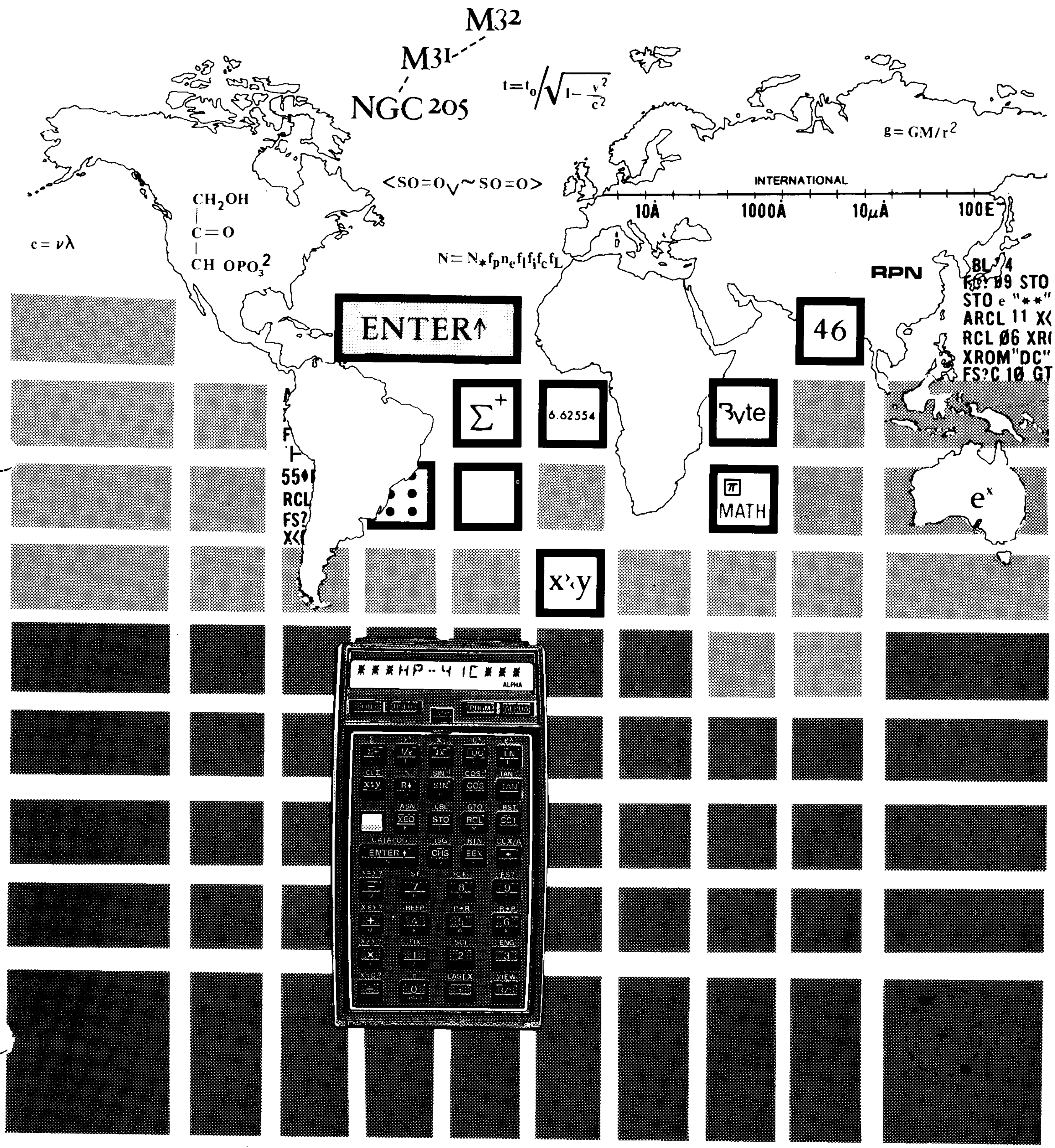
PPC ROM USER'S MANUAL

PRINTED IN U.S.A.



PPC ROM USER'S MANUAL

Dedicated to: Roger Hill

Whose intellect, enthusiasm, and profluent contributions are of incalculable value.

2545 W. CAMDEN PLACE
SANTA ANA, CA 92704
U.S.A.

FOREWORD

Because of the nature of the PPC ROM PROJECT, this manual is somewhat unusual. This manual is the effort of over one hundred users who worked directly on it, and many hundreds of others who indirectly contributed to its completion. Before diving into the routines, the PPC ROM user should first read the introductory material in Part 1, which includes the Preface, Organization and Use of Manual, Functional Grouping of Routines, Abstracts, and brief Introduction to Synthetic Programming. Once you have read Part 1 you may explore at random with a minimum of difficulty. Refer to the Glossary in the Appendices for definitions of unfamiliar terms.

This project is unique in the history of software projects. IBM and other large corporations have assigned multi-tens of programmers to a software project, but never before have over 100 programmers worked so long and so hard on a project--without compensation of any kind. The PPC ROM PROJECT is a community project in the true sense of the word. The project has always been completely public with month by month reports openly published for all to study and respond to.

It took two years and two months to complete. The first year was spent in mastering the HP-41 system, and while we were "first in line" for HP's announced Custom ROM Program, we waited until we could utilize the full power of the HP-41 to produce as complete a programmer's ROM as possible.

We believe in true personal computing and that a so-called higher level language is not always the path to greater computing power. We want to manage our always-too-small memory in ways we think are best. We prefer a flexible operating system that allows us to control our programming environment, and we want a well thought out operating system that can be altered if we wish. The routines in the PPC ROM express these interests and concerns. Much of the work that went into the ROM is original and makes a contribution to the Art. Here are a few examples.

- Programmed and documented by hundreds of users
- Outstanding ratio of features per byte
- Unusually complete technical details
- Personal contact for additional help
- A routines ROM - not an applications program ROM. This is a programmer's ROM.
- The full power of Synthetic Programming is made available to all HP-41 users.
- Operating system extension and enhancement programs
- Fastest known numerical sort routine
- Block and matrix operations defined and programmed
- Extended capability and improved accuracy in financial calculations
- Commendable integrator program
- Greatly expanded multiplot and high resolution graphics programs
- Matrix format printing of flags set in View Flags
- Skipping zero data in Block View
- Better access to all of HP's ROMs with **XE** Routine
- Expanded memory using **IP** and **PS** for QUAD "page" switching

One of the main objectives of the PPC ROM USER'S MANUAL is to provide an expression of the type of detail that programmers desire. This includes more than just a collection of general purpose routines with as many technical details as possible. The users are an essential part of the loop, and the PPC ROM project is designed to include user inputs. A portion of the ROM fund is being held in reserve for a follow-up addendum that will include:

a. Corrections for the errors found
b. Description of any BUGs that may be found
c. Additional examples
d. Additional Applications Programs
e. Suggestions for ROM or Manual improvement
f. Review of project
g. Conclusions and recommendations for future "user community" software development projects

A word about bugs. BUGs are of concern to all users. We define a BUG to be a failure of a routine or program to operate according to the complete instructions. Unless precise inputs and conditions are specified, you may have questions regarding the complete instructions. If you think you have found a BUG, we want to know about it. But first you should realize that after hundreds of hours of testing we haven't found any major BUGs. Therefore, a considerable effort on your part should be expended before you think BUG and call the PPC Clubhouse. Many "bugs" may be explained away by gaining a better understanding of the complete instructions. We do want to hear from you so your inputs may be included in the addendum. Happy BUG hunting.

There were many ideas for routines in the ROM that for various reasons never became a reality. It is possible that these creative ideas may appear in a future PPC ROM. We would like to have seen more alpha-string capabilities and diagnostic routines. In the math group we would like to have seen some routines in the statistics area. After reading this manual and mastering the PPC ROM, you will no doubt think of several routines that you will feel should also have been included.

We had planned special microcode routines that would have simultaneously simplified and expanded memory management, but the SDS system that would allow microcode in the ROM would have caused a three month delay, so these routines did not materialize. One reason alpha-string and diagnostic routines did not materialize was lack of space, and these kinds of routines tend to be memory intensive. There was very little discussion of statistics routines, and no specific statistics routines were actually submitted.

PREFACE

The PPC ROM project represents one of those rare occasions where a group of people join together to accomplish a work for the primary reason that it's a good idea. The ROM project wasn't undertaken to solve a common problem, nor was it accomplished to serve some commercial purpose. This manual represents the PPC ROM effort, but, most important, it is an expression of delight in the programming and application of a truly personal[1] computer. The PPC ROM project has been especially exciting to me because it partially implements an idea that dates back to early 1977.

In February 1977, I had the opportunity to visit National Semiconductor and spend two days "exercising" an EPROM version of their planned entry into the high-end personal programmable calculator market. The machine was the NS 7100. TI had not yet announced the TI-59 (it was announced in July of the same year). The 7100 was an exciting machine with such unique features as indirect addressing from all registers, 480 fully merged instruction capacity, and non-volatile file memory cartridges. I was most impressed with the attitude of the project managers because they wanted to give the 7100 user the ultimate in what would be called "...the world's first operating system based on a hand-held calculator."

After returning home I was thinking about National's new machine and the software that was envisioned for it. One Sunday Frank Vose (60) and I were discussing calculator functions. I had suggested that a calculator should have commonly used routines pre-programmed for the user to call in his programs to save program memory. Frank suggested a "list" of "needed" routines, and between us we had a substantial wish list. The routine concept began to grow in my mind, and on February 16, 1977, I wrote a ten page letter to National suggesting what I called a Routines Library. I envisioned a collection of routines that every machine owner would get. The owners manual would provide a wide range of programs that would essentially be a series of routine calls. Here is a quote from that letter (pages 2 & 3).

"The second, and main reason for this letter, is of such major importance, that I recommend that you review it immediately. As I have discussed with you before, and several others at National, I believe that the first Library cartridge produced should be a Routines Cartridge. The remainder of this letter will discuss this concept, its advantages, and provide a few suggestions for specific routines.

SOME IDEAS

NATIONAL ROUTINES LIBRARY NO. 1.

"The National Routines Library, NRL, offers the user a powerful capability of producing complete libraries of programs in many fields. By utilizing the routines of the 4096 - step NRL the user may write up to 32 (and more with special techniques) programs on a single file cartridge which acts as an executive program.

1. An alarm clock and a wristwatch are both considered personal timepieces. Only the wristwatch is a truly personal timepiece. Just like the personal timepiece, a personal computer is always with the user. A truly personal computer does not require a rigid form of use, such as sitting at a table. A truly personal computer is as easy to use while being pushed on a swing, as it is in the students' lecture hall.

This manual provides the instructions and program listing for hundreds of programs which the user may assemble in his own custom "library" in his own field(s) of interest. Specifically, executive routines are grouped in the following fields. (no special order)

1. Electrical Engineering
2. Chemical Engineering
3. Physical Science (Physics, etc.)
4. Civil Engineering
5. Finance and Business
6. Mechanical Engineering
7. Navigation and Aviation
8. Games
9. Numerical methods
10. Statistics

"The user simply selects the programs desired, assigns labels to them, and keys them from the listings in this manual. What makes this possible is the powerful set of generalized mathematical subroutines pre-programmed into the NRL. The short programs you key as an executive program may only be ten steps, but may execute many hundreds, or even thousands of steps of the NRL. In this way the user gets the best of two worlds - his programming, <u>and</u> the skills of the Pro's at National, as well, as a reasonable cost of writing ones own program. This concept fits well with the computer personality of the 7100 in that the user may work at a higher level than the usual calculator.

"This manual also provides complete descriptions of all the routines in the NRL including subroutine linking, accuracy and timing considerations."

TI announced the TI-59, and National decided not to produce the NS-7100. (About a dozen were actually made.) The routines concept stayed in my mind, and I "preached" the idea to any who would listen. On one occasion following a WESCON Calculator Session, I described the concept to the software manager at TI. He showed a strong interest, and I got the feeling that he almost grasped the concept. I would like to think that he was sufficiently "routine oriented" to use the concept in TI's Math/Utilities Library, but I will never know.

One "problem" with the routines concept--from the manufacturer's viewpoint--was the "purpose" of the routine. A program (or routine) must do a pre-determined job; it must be application oriented.

By the time the HP-41C was introduced in July 1979, I had reached another conclusion regarding the routines concept. The generality of the routines and their implications was a programming task that was far beyond the capabilities of any manufacturer. To do the job right, the whole user community should participate. What better approximation to the user community than PPC! In the August, 1979, *PPC JOURNAL* (V6N5P27c), I proposed that we take on a routines ROM. In late August of 1979 I sent a formal letter to Hewlett-Packard which stated in part:

"PPC would like to purchase an 8K HP-41C Custom ROM. Please consider this as a formal "Letter of Intent". Enclosed is a check (PPC #999) for $1,000.00 as a deposit."

The remaining part of the PPC ROM story has been recorded in the pages of the "Journal" in a dedicated ROM PROGRESS Column. The routines ROM that is described in this manual is not the ideal ROM of universal, pure function, routines originally envisioned. The 153 routines (a few are actually full blown applications programs, and a few are not routines at all) in the the PPC ROM, however, represent the creative efforts and talents of over 100 programmers and personal computing enthusiasts. It is an accomplishment beyond any dreams (of reality) that I may have had.

Now that the PPC ROM Project is history, it is appropriate to ask: "What was the most difficult aspect of the project?" Aside from the unplanned growth (from a 150-200 page manual for 500 users to this tome for 2500-and-more future users) I personally will remember two areas of difficulty. The first involves project management and also personally doing a portion of the ROM. John Kennedy (918) lifted the burden of 50% of my routines (see PPC CJ,V8N1P10a), and this made it possible for me to handle the Housekeeping Group. Actually, working on the routines is what makes it worth the effort, so I would never want to be "demoted" to only the management aspect of such a project.

The second "problem" I will always remember regarding this project is the "order processing" aspect--the bookkeeping, logging of orders, etc. No matter how clear you make the instructions, a significant percentage of the participants will expect you to do something special for them. My recommendations for anyone trying to do something of this magnitude is to carefully outline the order process and simply return any order that is not in conformance. You don't have to be so harsh if you have adequate man power or time. I had neither and learned from the experience.

Any project involving hundreds of people is a challenge. As any manager knows, there are no unsolvable technical problems; all "problems" are people problems. In this regard I am amazed that we didn't have the classical problems of fighting over what would be done, how, and by whom. If we had a problem, it was the spouse who couldn't understand why so much time was spent "on that PPC thing." A volunteer activity must continuously live with "IT'S 2 A.M.; I've got to get home or my wife will divorce Me." Of course, employers "don't understand" either, but somehow we all managed to squeeze in the necessary hours to become a part of the most exciting software project ever undertaken by a user group.

Much of the success of the project was due to the dedication of the four ROM committee members. Jake Schwartz (1820) and John Kennedy (918) had been involved with PPC Projects before and had some faint idea of what effort might be required. Keith Jarett (4360), however, sort of stumbled into the project, and by all rights should have thrown up his hands and said, "You are all mad." I will never forget the early morning SDS loading session when I said, "It's your turn." He looked at me and said, "Never in my life have I worked more than 24 hours straight on any project." We had an unofficial fifth committee member, Roger Hill (4940). Roger has spent hundreds and hundreds of hours of his time programming, debugging, and documenting ROM routines. His mark on the synthetic group is bold and bright. He has literally contributed ideas and programs to all four groups. It was obvious to all committee members that we dedicate the *PPC ROM USER'S MANUAL* to him. I am sure that all 4,000 PPC members want to say, thank you Roger.

The number of man hours spent in perfecting and documenting the PPC ROM routines is impossible to record. Ray Evans (4928) conservatively estimates 400 hours for his **S2** routine. His effort was measured against the best. How many hours were previously spent by others on sort programs? The ROM Progress Columns from August 1979 to July 1981 contained 70½ pages of ROM related topics. This is equivalent to 140,000 words. I can account for at least ten man years (20,000 hours) of effort, and I'll bet that if the total number of man hours that PPC members have spent programming, studying, testing, and thinking about ROM routines and topics were considered, the time would approach one man century. I only wish we had an extra 200 hours to better integrate the routines and an extra 300 hours to produce a better manual.

It is true that PPC members have mastered the HP-41 all by themselves. It is also true, however, that while those wild and weird synthetic programmers were exploring the 41 system, HP was quietly cheering and applauding their effort. It is not possible for HP to endorse or approve synthetic instructions. Also, there was no reason to restrict these instructions from being placed into the ROM--at PPC's risk, of course. Formally, informally, officially, and unofficially, HP personal assisted where they could. Specially prepared short programs were loaded and listed to test the SDS System. Today we know that the risk is very low, when we committed to a $20^{+}\%$ synthetic-instruction ROM; however, we really didn't know.

It is difficult remember now, but the proposed ROM was 98% guts and 2% knowledge in late 1979. Today the percentages are reversed. Synthetic programming is a rapidly maturing activity and the PPC ROM will bring all its power to the user community. HP's contribution, in a hundred ways, was climaxed on August 22, 1981, a week shy of two years from the day I mailed the letter of intent. HP made a special effort to provide 100 ROM's for the PPC Northwest Conference at Oregon State University in Corvallis. The conference attendees were the first group to get hands-on experience with the ROM.

We cannot give credit to every person for every detail of effort on this project. Special mention must be made, however, of the Orange County-Los Angeles members who did much of the work in the final paste-up, packaging, and shipping of the 30,000 pounds of ROM's and ROM Manuals. These 'local' members are the same members who stuff, seal, and mail the 'Journal' each month. We have tried to list all known contributors in Appendice B.

If you are a PPC member you can be proud to have been a part of this project.

Happy Programming.
Richard J. Nelson

I'm not sure of the exact date when Richard Nelson first discussed with me the idea of forming a set of programming routines for the HP-41C, but the time frame was a month or two prior to August 1979. For the last two years the PPC ROM project has consumed a considerable portion of my spare time.

I am honored to have served on the PPC ROM committee and hope that others may profit from the contributions made by all those involved. In particular I would like to thank Graeme Dennes (1757), Don Dewey (5148), Phi Trinh (6171), Read Predmore (5184), Roger Hill (4940), George Eldridge (5575), Richard Schwartz (2289), and Bill Barnett (1514) for their special contributions.

Many others are also deserving of thanks, but I consider the contributions made by these people to be significant.

The history of the project is somewhat documented in the ROM PROGRESS column of the 'PPC Journal'. But what is missing is an indication of the effort required to bring the project to its conclusion. An estimate of the number of man years would probably fall short of the true number. Perhaps even more significant to those of us intimately involved in the project has been the weight of the responsibilities on our psyches.

I can clearly remember a feeling of relief when the time came to ship the disk to HP that contained the accumulated programs for the ROM. This marked the completion of one phase of the project. My responsibilities were for the math routines, and overall I felt the size and variety of routines fit together fairly well, but it took a long time to bring each program to its final form. For each routine I kept a current listing with a complete stack trace. This meant that for each little change that was made I had to completely re-write parts of the documentation that I was keeping. Each new change was more agonizing, and I had the feeling that nothing was permanent.

There were hundreds of changes made, and I felt like an artist who could not finish a painting. There was always some improvement to be considered that made completing the job impossible. In the end, it was a relief to know our work would be cast in concrete and that no more changes could be accomodated.

After the disk was shipped, there were some last minute changes, but soon it was truly all over. Nothing more could be added and nothing would be taken away. A short period of time passed, and then we faced a decision of whether to wait for a new SDS system that might allow us to include the planned special microcode routines in the ROM. This would have caused another 3-month delay and there was no guarantee the new SDS system would accomodate our special routines.

The ROM Committee's decision was unanimous that we not delay any longer. We really were committed now. The disk was accepted and the masks were made.

The next phase of the project was work on the manual documentation. Everyone involved in programming agrees that the work required to document programs is considerably greater than the programming effort that produces the programs. Although I never doubted this, I now <u>know</u> that it is true. The documentation went through many changes, but I was now using a word processor and the changes seemed less painful. It still took a full day to print the 280-plus pages that I accumulated, and I went through several printings.

All of my documentation (33 programs) was kept on three 8-inch floppy diskettes. When it came time to make the final printing, I needed to get a Dual Gothic printwheel for my printer. I literally called every word processing supplier in the Los Angeles area and could not locate that particular element. I began to wonder if I was ever going to get 9 months work off those diskettes and onto paper.

No one that I called had the printwheel in stock and no one was able to locate one when they called their suppliers. Calls were made all across the country and Richard Nelson finally located a dealer in Massachusetts who had the printwheel in stock. When the printwheel arrived the shipping case was cracked, but the precious element wasn't damaged and I was able to make the final printing. Use of a new waxing machine required that I feed the paper to my printer one sheet at a time. By the end of the day I had blisters on my fingers, but I was relieved that my portion of the documentation was nearing final form.

One theme that kept recurring in the whole project was that, when you reached the point at which you thought you were done with something, you were really only about half-way done. Everything took twice as long to do compared to the best estimates made by rational human beings. As I collected my printed sheets, little did I know how much work remained.

It was after 5 A.M. in the morning when Richard Nelson and I finished cutting my material. Thanks to a new waxing machine, the time required to wax all the cut pieces for paste-up was much shorter than it would have been otherwise. As I left the clubhouse to go home and assemble all the work, Richard told me it would probably take one hour to lay out 6 manual pages. I expected to do 20 pages in one hour, but in fact, it took three times as long to complete that particular task. I ended up with 125 pages of material. At times it seemed endless.

The next job was to lay in all the special typeset titles and special symbols and the two-character global labels. This seemingly simple task occupied one full week of my summer vacation. But I could see things fitting together and became more enthusiastic about finishing the manual. There was more proof reading to be done and then corrections had to be made, and some final additions were made. All little things, but all time consuming and always on your mind. We were producing documentation as well as artwork, and both jobs are perhaps little understood by programmers. The programming done earlier was definitely easier than producing the documentation and artwork.

The whole project seemed to grow as progress in each of the phases was completed. The task before us has turned out to be rather enormous, and I'm not convinced we could repeat our accomplishments. Two years ago I did not expect the final outcome to be as it is now. The quantity and quality is more than I should have expected.

At the recent Corvallis, Oregon, conference people were discussing "the next" PPC ROM, but I know that there are a few people (to go nameless for sure) who will never again volunteer for a project like this. There were times when each of us would have been satisfied to have quit before the final goal was accomplished.

The rewards will come as all of us learn from the information provided and as we develop new programs and new programming techniques. But a word of warning should go out to those members contemplating doing another PPC ROM. DON'T DO IT! Just enjoy the ROM you have. You really don't want to know how much work is involved once you start such a project.

John Kennedy (918)
PPC ROM Committee Member

The purpose of the Peripheral Routines section of the PPC ROM is to extend the capabilities of HP-41 peripheral devices. For the wand, the <u>B</u>arcode <u>A</u>nalyzer program is an analog of the card reader's $\overline{\text{v}}$erify operation. For the 82143A printer, three areas are concentrated upon for enhancement. First, to allow formatted columns of printed information, <u>C</u>olumnar <u>P</u>rint Formatting is presented. Secondly, $\overline{\text{t}}$o enhance the printer's plotting capabilities, we have the High Resolution Histogram/ Histogram with Axis and the Multifunction Plotting/ High Resolution Plotting pairs of routines. Thirdly, to significantly expand the printer's character set, the Special Characters routine was proposed for the ROM. This was later deleted; however, it appears in its entirety in barcode and with complete instructions in this manual in Appendices M and L respectively.

It is my feeling that the Peripheral Routines section achieves its goal of expanding the usefulness of the wand and printer. So much good material was received for proposed inclusion in this section of the PPC ROM that eventually a custom module dedicated to peripheral routines may be justified. Perhaps this ROM could be written in assembly language? More peripherals in the future should also prove the need for such a module. However, for now we may look forward to new prospects for better programming with the PPC ROM as we know it, in conjunction with the HP-41C/V personal computing system.

For the sake of completeness and to acknowledge the fine efforts by PPC members everywhere, here is a (partial?) list of routines, complete or under development, which were considered for inclusion in the peripheral routines section:

1. 3-D Grey Scale Plotting	Steve Wandzura (4635)
2. Y=f(X) Function Value Tabular Printing	John Dearing (2791)
3. Text Justifier Program	Roger Hill (4940)
4. Family of Curves Plotting	William Wimsatt (5807)
5. 3-D Plotting	Valentin Albillo (4747)
6. More Special Characters (thousands!!)	Many members
7. Generallized HP-41C (X,Y) Graphics Plotting	John Burkhart (4382)

There are many people to thank for their assistance in completing this project, some of whom deserve special recognition. There's Tim Fischer (5793), who saved us over 300 bytes in both combining and greatly improving the multifunction and high resolution plotting programs. And, of course, Roger Hill (4940), who seemingly had a hand in cleaning up every routine in every section, and for programming 'under pressure' whenever the need arose. Many thanks to Richard Schwartz (2289) for making sure the peripheral routines made it safely through the SDS System with all updates and corrections intact. My good feelings also go out to those who provided miles of printer paper and pounds for magnetic cards for the cause; in alphabetical order: George Duba (4248), Jerry Lee (5406), Charles Slocum (2907) and Jack Sutton (5622). Last, I'd like to thank the ten or so people from the Philadelphia Chapter who aided in writing, editing, programming, debugging, etc. until the manual was done. Special thanks to Mark Trebing (4421), who donated time and effort to type many pages into a word processor, and to Charles Allen (4691) for extensive testing and documentation right up to the last minute. We're all better off as a result of the volunteer work these few have provided.

Jake Schwartz (1820)
PPC ROM Committee Member

The ROM has been a bigger project than I ever anticipated, primarily because of the extensive documentation that Richard wanted. I think that you'll find the documentation quite complete, especially considering the short amount of time and the fact that most of the people involved had full time jobs to contend with. One of the major surprises was that except for some of the more complex routines, there was very little overlap between the group of members that submitted synthetic programs and the group that worked on the writeups.

I'd like to thank everyone who helped out, especially those who did the unglamourous work of documentation. Those who contributed programs, whether or not they were the final version, are generally credited in the Contributor's History for each routine. Errors and omissions, of which there are undoubtedly several, will be corrected in the addendum if anyone notifies me. The people that I'd like to thank for writeups are Roger Hill (4940) for MK, 1K, +K, PK, LB, L-, and -B; Harry Bertuccelli (3994) for CU, CX, ΣC, HD, UD, The Towers of Hanoi Appendix (APPENDIX A), PPC ROM Pocket Guide, and most of the Curtain Moving Appendix (APPENDIX M); David E. White (5353) for DT, RF, SD, RD, SK, RK, and rough draft material for ML, S?, C?, and Σ?; Greg McCurdy (3957) for NR, NS, E?, 2D, CK, and OM; Les Matson (5608) for XE, IF, and RT: Richard H. Hall (4803) for NH and HN; Tom Cadwallader (3502) for Ab, Sb, Rb, and VK; Paul Lind (6157) for LR and SR; Dave R. Kaplan (3678) for MT, EX, and VM; Carter Buck (4783) for NC and SU; William Cheeseman (4381) for AL; Keith Kendall (5425) and Doug Fauser (4968) for rough draft material for PD, DP, and CB; and Joe Bell (5781) for rough draft material for IP and PS. I am grateful to Bill Wickes (3735) who wrote the Introduction to Synthetic Programming, and to John McGechie (3324) who wrote the early history of MK. I'd also like to thank Charles Ragsdale (7251) for writing virtually all of the synthetic routine abstracts as well as rough draft material for XD. Last, but most certainly not least, I thank George Duba (4248) and Clifford Stern (4516) for their compilation of the technical details tables.

Now that this thing is done, I think I'll call Jack Baldrige's desert island travel agency...

Keith Jarett (4360)
PPC ROM Committee Member

The PPC ROM Committee was composed of the following people.

Math Routines - John R. Kennedy (918), Math teacher at Santa Monica College in Santa Monica, California.

Peripheral Routines - Jake Schwartz (1820), Bio-Medical Engineer at Childrens Hopital, Philadelphia, Pennsylvania.

Synthetic Routines, Keith Jarett (4360), Systems Engineer, at Hughes Aircraft, El Segundo, California.

Housekeeping Routines, Richard J. Nelson (1), Electronics Engineer/Consultant, Santa Ana, California.

All ROM inputs were divided into one of the four groups managed by the group coordinator. The 122 global-labeled routines are listed below under their group coordinator.

ROM Routines Listed by Documentation Coordinator:

Keith Jarett - (Synthetic Routines)

+K	CU	HN	NR	RX	VK
-B	CX	IF	NS	Rb	VM
1K	DC	IP	OM	S?	VS
2D	DP	L-	PA	SD	XD
A?	DS	LB	PD	SK	XE
AD	DT	LF	PK	SR	Σ?
AL	E?	LR	PS	SU	ΣC
Ab	EP	MK	QR	SX	
C?	EX	ML	RD	Sb	
CB	F?	MT	RF	UD	
CD	GE	NC	RK	VA	
CK	HD	NH	RT	VF	(67)

Jake Schwartz - (Peripheral Routines)

BA	HA	HS	MP	
CP	HP	LG		(7)

Richard Nelson - (Housekeeping Routines)

AM	BV	MS	S2	T1	
BI	FL	PO	S3	TN	
BL	MA	S1	SM	XL	(15)

John Kennedy - (Math Routines)

BC *	BΣ *	DR *	IR *	M5 *	SE *
BD	CA	FD	JC	NP	TB
BE *	CJ	FI	M1 *	PM	UR *
BM *	CM	FR	M2 *	PR *	
BR *	CV	GN	M3 *	RN	(16*)
BX *	DF	IG	M4 *	SV	(33)

*These routines are actually part of the Housekeeping group. John Kennedy assumed responsibility for these routines.

P R E F A C E   P O S T S C R I P T

In addition to this manual the PPC ROM has three additional items for making the programmers task a little easier. These are:

a. PPC ROM Pocket Guide

b. Plastic HEX Table

c. Optional, more bold label

Each is self explanatory. The plastic card is conveniently kept in the Pocket Guide.

The *PPC ROM USER'S MANUAL* was originally planned to be of three distinct parts as described in the Organization and Use of Manual section beginning on page 1. Part I was to be the introductory and explanatory material. Part II was to be the main body of the manual with references grouped in Appendices as Part III. When the manual was assembled, however, there was no room for most of Part III if the 500 page budget was to be maintained. When 10,000 copies of a manual are printed, adding "a few more pages" can "add many more dollars" so the organization of Part III had to be changed.

The problems of assembling the art work from four sources dictated that a plan of pagination be adopted that would allow concurrent paste-up of the routines. For this reason, and the desire to provide rear tab indixing for the 122 routines, it was decided to start each routine on an even page. If a given routine happened to have an odd number of pages there would be a few odd pages sprinkled throughout the manual. As it turned out, there were 39 such pages. The original plan was to fill these pages with related reference materials such as forms, tables, artwork, etc. An alternate plan was to provide a formal NOTES page that was identified with space in the index for the user to fill in for his own reference.

The solution to the problem was to use the "odd" pages for the "leftover" part three material. This solution was essential, because we had referenced the appendices in previously pasted-up parts of the manual and it would have been too much work to redo the manual at the last minute. This was especially important, because the PPC communication link was at stand still while this manual was being finished. The phone bulletin was receiving calls at the rate of 20 per hour around the clock. Members were expecting their ROM's.

The Appendices are scattered, in order, throughout the manual on pages 33, 37, 61, 105, 109, 121, 131, 135, 143, 165, 169, 173, 187, 227, 233, 251, 259, 273, 297, 335, 345, 349, 367, 373, 387, 415, 423, 429, 439, 443, 449, and the 'normal' continuous pages of 466 thru 487. Most of the broken up Appendices are single page, but a few are five pages in length. These are started with an "Appendix X from Page N" and end with "Appendix X Continued on Page N". As an additional aid in using the "misplaced" Appendices we have boxed their page numbers. They are all on odd pages and you may use this designation to turn the pages to the next Appendix.

This unconventional organization was essential if the 500 page budget was to be maintained and no material was to be left out. We accomplished both objectives.

None of the PPC ROM objectives could have been met without the encouragement and support of the PPC membership at large. For me, the time period between the first two U.S. Space Shuttle launches was one of intense concentration on this manual. Without the infinite patience of my wife Paz, none of my PPC activities would be possible, and this manual would have a 1983 date on it.

Richard Nelson

TABLE OF CONTENTS

PART I - INTRODUCTORY MATERIAL

PART II - ROM ROUTINE INSTRUCTIONS

PART III - SUPPORTATIVE MATERIAL (APPENDICES)

ORGANIZATION AND USE OF MANUAL

This manual is divided into three major parts. Part I contains the introductory reading material. Part II is the working (user's) part of the manual. Part III contains the reference and resource materials.

Part I consists of introductory material. This includes the foreword, the prefaces, the table of contents, the section which you are now reading which describes the organization and explains the use of the manual, a table of the functional grouping of the routines, the abstracts of all 122 global labels, and a section which is a brief introduction to synthetic programming.

Part II consists of the complete write-ups for each of the 122 global labels and is really the heart of the entire manual. The material contained in Part I and Part III simply augments and supports the complete descriptions of all the routines which appear in alphabetical order in Part II.

Part III contains the appendices and the index. The titles of the appendices include: Advanced Applications of LR/SR & HD/UD; ROM PROJECT Contributors; ROM Routine Author List; References and Accessories (commercial products); ROM PROJECT Expense Summary; ROM Order List; Glossary of Terms; Table of Tables; Illustrations & Figures; ROM Listing; Routine Label-XROM Table; Special Characters; Barcodes of ROM Routines; Barcodes of Applications Programs.

The table consisting of the functional grouping of the ROM routines helps place together those routines that are logically related by function. The categories or groups consist of: alpha register usage, block operations, curtain operations, display functions, key assignments, loading bytes, general mathematics, matrix operations, memory management, miscellaneous, non-normalized numbers, peripherals, program pointer, return stack, and sorting operations.

Thus all 122 routines have been grouped into 15 categories that provide a broad outline of the ROM. Since there is no agreement on the exact placement into the categories, any routine may appear in more than one group. The grouping table will make it faster and easier to determine routines related by function. Those readers with special interests may wish to initially concentrate on one particular group. The functional grouping table makes it easy to determine a desired two-character label, but the alphabetical order of the individual routine write-ups in Part II makes it easier to actually locate within the manual the complete information about the desired routine.

The list of abstracts which appears with the other introductory material in Part I is provided for those readers who on their first reading wish to obtain an overview of the features and capabilities of the ROM routines. The abstracts of all 122 routines are collected in alphabetical order and located in one place.

Each routine write-up in Part II conforms to the following format. Note that many of the major sections are optional, but, when these sections are present, they will appear in the following order:

Title
Abstract
Short Example
Background (optional)
Complete Instructions
More Examples
Further Discussion (optional)
Application Programs (optional)
Formulas Used (optional)
Routine Listing (form provided)
Line By Line Analysis
References (optional)
Contributor's History
Final Remarks (optional)
Further Assistance
Notes (optional, form provided)
Technical Details (form provided)

Each routine begins with its title at the top of an even numbered page and contains in reverse print the two characters used in the global label. This is immediately followed by a one- or two-paragraph abstract which describes (without technical details) the purpose and function of the routine. ROM routines are referenced throughout the manual by two-character reverse print symbols.

Next appear one or two short examples which illustrate what is required to run the routine. Although these first examples are called "short", they are not necessarily short in length, but rather illustrate a typical use of the routine. The purpose of these first examples is future reference on how to run the routine. They provide the user who is vaguely familiar with the routine an illustration of its use which avoids a long, detailed reading of the complete instructions.

The next section is called Background and is optional since many of the routines are simple in nature and generally do not require any special background knowledge. When a background section occurs, it provides a transition between the abstract and first examples and the complete instructions.

The Complete Instructions section is self-explanatory and is the primary reference point when you have a question about how to run the routine. These instructions will describe step by step how to exercise the routine. The Complete Instructions section is then followed by more examples which further illustrate the use and options associated with the routine.

Some routines may benefit from further discussion following the complete instructions and the examples. The information provided under Further Discussion is not necessarily required to run the routine, but may provide insight on how to get the most out of the routine.

Applications Programs are programs which use or call the global label as a subroutine. These programs will illustrate and provide ideas for how you may use the routine in your own programs. There are many creative applications that we either did not think of or did not have time to fully develop. It is expected that you will also develop and share with other PPC members your own applications programs. Not all routines have complete applications programs.

Many of the math routines contain formulas, and these are listed for reference purposes, in some cases with comments about their implementation. This section is followed by the Routine Listing which apears in a special boxed form. Although the majority of routines are straightforward, a few of the more complex synthetic routines will not include listings of all the called subroutines. Where convenient, the called subroutine lines are given in addition to the program lines that make up the routine. Don't be surprised when you see two sets of line numbers that are not consecutive.

The Line By Line Analysis section follows the routine listing so that line numbers are easily referred to. This part will describe in some detail exactly how the various program lines accomplish the objective of each routine. Those interested in this detailed form of documentation will find many enjoyable hours of reading and studying the programming techniques that are to be found here.

References are optional and may refer to previous PPC Journal articles, books, or other periodicals. You may wish to add your own references in some cases. The Contributor's History is a brief indication of the people instrumental in making the routine a reality. The Final Remarks section is provided as a starting point for future work. The comments here may apply to future dreams about how the routine could be improved if the routine were to be implemented on a future machine, especially if we assume such a machine has few constraints.

As if all the above were not enough, we have also tried to provide the names of two people you may contact for further assistance on the routine. You should feel free to contact these people if you have difficulty making the routine behave as described. These people may also wish to hear your feedback on the routines. At the end of some of the routines a little space was left over, so we provided a form on which you may write your own notes that should contain everything we left out.

The technical details for each routine are provided in a special table which shows registers and flags used, local labels and global labels called, execution times, display and angle modes and other technical details. The top of the table shows the XROM numbers and the SIZE require to run the routine. In many cases the indicated SIZE is a minimum and in other cases the SIZE is given as variable or no special SIZE is required.

All the stack and data registers used are listed. Where possible the content of registers used is briefly indicated, but in some cases the values are not constant so only the word "used" is given. The same is true of flags used. Where possible we have tried to indicate the significance of the flag being set or cleared. In some cases more complete information will have to be obtained from the Complete Instructions section. The Technical Details section is primarily intended to be used for quick reference.

The Unused Subroutine Levels is the number of remaining levels when the routine is called as a direct subroutine from one of your own programs. Global labels called may be direct or secondary. A secondary call occurs when the ROM subroutine calls another ROM subroutine. Some global labels are used with a GTO or are dropped into by another routine and hence do not appear as direct subroutine calls.

The term "interruptible" near the bottom of the Technical Details table means that you may press R/S twice: once to stop the routine and a second time to resume execution; no detrimental effects will result. Except for special synthetics, most routines are interruptible. The meaning of Execute Anytime is that the routine requires no data input and that no special condition of the machine is required. All that needs to be done is to execute the two-character global label. Only a few of the routines fit in this category.

If you load a routine into RAM, you may find a discrepancy between the actual byte count and the number given in the table. In many cases, the byte count in the table may be reduced because of RTN/END combinations or extra local labels or other parts not always required. Because the HP-41C COPY function only applies to complete program files, the number of registers to copy is the number of registers for the entire file.

The brief introduction to synthetic programming is provided for beginners who are not familiar with synthetic functions but may also serve as review for experienced synthetic programmers. The techniques and applications of synthetic programming are ever expanding, so a complete coverage of this topic is neither intended nor possible in this part of the manual. This introduction contains a description of the functions of the internal status registers and includes a memory map of the HP-41 system. An HP-41 Combined Hex/Decimal Byte Table and its use to create synthetic instructions are explained. More information about synthetic programming may be obtained from the references. This introduction serves the purpose of helping novice users obtain a minimal background for using the synthetic routines that are in the ROM.

Part III of the manual contains the appendices and the index. The index is self-explanatory.

Appendix A illustrates how expanded operating system concepts advocated by PPC members may be implemented by synthetic programming. Hundreds of people have contributed to this project.

Appendix B lists the PPC members who have made signifcant contributions. Our apologies for any omissions, which we will correct in the addendum to be produced in mid 1982. Please contact us if your contribution is not listed.

Appendix C lists those persons who may be considered the primary authors of routines. The original intention was to assign credit to one person per routine, but in some cases more than one name is included, and in other cases it was not possible to assign credit due to the many people who contributed.

Appendix D lists some commercial products related to the HP-41 and provides references for further information about these products. Included are a port extender, EPROM boxes, magnetic card holders, books and other accesories.

Appendix E contains the summary of the expenditures for this project. Because the ROM is a custom product which has a substantial tooling charge, this project could only have been accomplished by having members pool their resources and purchase the ROM as a group. The PPC ROM PROJECT is a special project of PPC.

Appendix F contains the list of the ROM orders by member number. 5000 ROMs were purchased from Hewlett-Packard for this project. Each order received is for two ROMs including manuals. 2500 orders comprise the complete project. The ROM order list accounts for these 2500 orders and is arranged by member number order.

Appendix G contains a glossary of technical terms. We discourage unnecessary jargon, but the study of personal computers necessitates technical terminology for effective communication. We wish to promote consistent use of standard terminology and this includes terms developed by HP and PPC.

Appendix H lists any tabular data found in the manual.

Appendix I lists illustrations and figures found in the manual.

Appendix J is a complete line-by-line listing of the entire PPC ROM. This listing was done in the NORMAL mode on the 82143A Printer because non-printable ASCII characters are more easily observed in this mode. The spacing in NORMAL mode also makes it easier to locate labels.

Appendix K lists XROM numbers and corresponding function labels for the PPC ROM, the 82143A Printer and the Card Reader. Two lists are provided. One is in XROM number order (this is the same as CATALOG order) and the other is alphabetical order by function name.

Appendix L contains a program originally planned for the ROM. At the time of final selection and loading it was decided that these 1000+ bytes would be better utilized if replaced by many shorter routines. This special characters routine is described in the same format as the other ROM routines. It is in barcode, and it includes a demonstration program for the printer.

Appendix M contains introductory and background material on curtain moving (re-numbering of data registers under program control). The characteristics and usage of the PPC ROM curtain moving routines CU, CX, HD, UD, and ΣC are compared.

Appendix N contains the 22 program files that comprise the ROM and is printed on special paper near the end of the manual. These barcodes correspond exactly with the ROM listing in Appendix J. See paragraphs below for details on copying and/or loading ROM routines into RAM.

Appendix N contains barcodes of the more signifcant applications programs that are described in the individual routine write-ups. Many of these routines contain synthetic instructions.

A few remarks concerning some of the routines are included here because these notes did not fit as part of any one particular routine's documentation. There are six "keyboards" associated with the math routines. Four of these keyboards are fully documented in the routine write-ups for CA, CV, FI, and FR. However, two program files, BD and IG, also contain local labels associated with the top row of keys. If you key GTO "BD" and switch on USER mode, the following routines will be assigned (default) to keys A, B, C, D, E, e:

				JC
BD	TB	PM	CM	CJ

If you key GTO "IG" and switch on USER mode, the following routines will be assigned (default) to keys B, C, and D:

	IG	SV	FD	

By using the top row of keys you can avoid keying XEQ "XX" where XX is one of the associated global labels. There is no mention of this use of these keys in the write-ups for any of these global labels. For example, when you first exercise the examples in the base conversion routines BD and TB, you may find it more convenient to stop the program pointer in the program file BD (key GTO "BD") and then simply press A or B to execute BD or TB.

As another example, pressing C in USER mode, when in the file IG, will execute SV, the solve routine (assuming of course no other function has been assigned to key C). These local labels were first used when the routines were under development, and it was decided that it would be a convenience feature to leave the local labels in the final ROM version.

The following application programs of math routines are in barcode at the end of the manual, and no mention of this resource is made in the corresponding routine write-ups.

CVPL (CV)
LPAS and FAST (FI)
MIO and RRM (M1)
PHN (NP)

Another remark may apply to any one of the 122 global labels. In RAM program memory you can usually simulate, alter, or customize a global label to your own liking. This may be desirable if your application program makes many calls of a ROM routine, especially if that routine requires a certain setup before it can be called. Most users would probably do this anyway, but there may be times when you feel a ROM routine prevents you from using it the way you feel it should be used. In almost all cases, a short customized setup before a call to a ROM global label will free you from apparent limitations.

As a simple example, someone is bound to question why the base conversion routines are called BD and TB. If these two routines are inverses of each other, why weren't the "idiots" who programmed them consistent in their selection of global labels? It would have been more consistent to name them BD and DB or TB and BT.

The reason the obvious choices were not used was to avoid conflicts with other global labels that are used by HP in other existing ROMs. See V7N10P7b for a list of two-letter labels used by HP as of early 1981. But in any case, if you don't use those ROMs you could write the program:

```
01 LBL*BT
02 XROM BD
03 RTN
```

which gives you the combination TB and BT which you'll have an easier time remembering.

As another simple example, the input for the block rotate routine BR is assumed to be in the form:

Y: 1st register in source block
X: ± number of registers within the block

If you would prefer to enter the block parameters in the form ±bbb.eee (the sign of X determines the direction of rotation), the following customized routine may be used: BRX = Block Rotate Extended

```
01 LBL*BRX      09 E-3
02 SIGN         10 ST + Y
03 ST * L       11 /
04 LASTX        12 R↑
05 ENTER↑       13 -
06 INT          14 *
07 STO T        15 GTO BR
08 -
```

The PPC ROM is an extensive, complex piece of software designed to serve many users in diverse areas. Because of this aspect of the ROM, users in different areas may find apparent inconsistencies. There were many implementation decisions made as the ROM was being developed. While you may not agree with (or may be ignorant of) the decisions made, it is usually possible to work around apparent obstacles, and by doing so you'll find more use for and have a greater appreciation of the PPC ROM.

A few notes follow concerning copying ROM routines. The PPC ROM is comprised of 8,170 bytes (22 bytes unused) programmed in 22 files--remember that an HP-41 program file is that series of instructions between two "END" instructions. The ordering of the routines is for maximum use of ROM bytes. Shared code and the falling of one routine into another, while maximizing the number of routines the ROM may hold, tends to confuse the user, because of an apparent lack of order. The routine ordering also makes it difficult to combine routines into a common file so that they may be run in RAM. Caution must be exercised if you copy ROM routines into RAM and expect them to run correctly.

1. Copy all called or used routines.
2. Convert all XROM calls to XEQ.
3. Change all local labels that are duplicated and have "reverse" calls to them. This is most important when deleting ENDs to combine routines into functional blocks.
4. Identify--as a suggested convention--RAM versions of PPC ROM routines with three-letter global labels having the first two letters the same as the ROM and the third a duplicate of the second.
5. RAM versions will usually run 10 to 25% slower than ROM versions.
6. EP and PS are routines that are required to be in ROM; they will not run in RAM without major changes or special operating conditions.

The optimization of label length (number of characters) and the number of returns dictated that file lengths be 63 registers or less so as to fit into a basic HP-41 without memory modules. The introduction of the HP-41C/CV and Quad Memory Module made this requirement less important. Two program files exceed this requirement (LB - 71 registers and MP- 88 registers).

The synthetic ROM routines provide capabilities never before seen in a programmable calculator. These capabilities range from convenience features like SIZE finding to operating system enhancements like key-assignment management and curtain control (programmable register renumbering). The large number of synthetic routines and the unforgiving nature of a few of them may be intimidating, or at least confusing, to the beginner. Therefore an attempt has been made to group the synthetic routines into three levels of difficulty, corresponding to the amount of knowledge needed to make real use of them.

The Level I routines can be used by anyone who has read the Owner's Handbook. They require no knowledge of the internal workings of the machine. Many of the Level II routines require a rudimentary knowledge of HP-41C memory structure or program instruction structure (the byte table). In most cases, however, the needed information is contained in the routine write-up. Level III routines require more background reading to use them to their full potential. For example, there is a separate appendix discussing curtain moving, and the subroutine stack extension program LR has a lengthy write-up with several application programs.

The Technical Details tables for the synthetic routines use a slightly different convention for stack usage than do the other routines. The contents of the stack _after_ execution of the routine are listed.

Level I	Level II	Level III
A?	+K	-B
AD	1K	2D
CK	AL	Ab
DS	BL	CX *
DT	C?	CU *
EX	CB	DP
F?	CD	HD
GE	DC	L-
MT	E?	LF
NR	EP	LR
NS	FL	OM *
PK	HN	PD
QR	IF	RX
RD	IP	Rb
RF	LB	RT
RK	MK	RX *
S?	ML *	SR
SD	NC	SX *
SK	NH	Sb
T1	PA	UD *
TN	PS	ΣC
VA	SU	
VF	XD	
VK	XE	
VM	XL	
VS		
Σ?		

*significant risk of MEMORY LOST with improper use

LF and all the routines that call **LF** replace register contents of all registers between the last key assignment and the .END. with zero. Be aware of this if you have any data in these registers due to your programming or HP's ROMs or accessories. This means that **MK**, **1K**, **+K**, etc. will overwrite these registers.

Another note of caution: DON'T R/S INDISCRIMINATELY IN THE PPC ROM. Very nasty things can happen if you don't watch what you're doing. For example, if you R/S after **VA** or **TN**, you get MEMORY LOST (usually), courtesy of **UD** or **CX**. At the very least, an indiscriminate R/S is likely to cause your flag register contents to be disrupted.

The synthetic routines are much more than just a collection of programs; they are a fully integrated package. For example, there are 68 XROM instructions (calls of other PPC ROM routines) in the synthetic group. This amount of repeated use of routines saves many bytes, enabling more and better routines to be in the available ROM space.

The PPC ROM is like a high-performance sports car. It can run circles around its ordinary cousins, but it demands more skill of its operator and is less forgiving of mistakes. If you keep this in mind you'll find that the PPC ROM is a lot of fun to drive.

NOTES

ROM ROUTINES GROUPED BY FUNCTION

LBL..............is two letter global label.
ROUTINE TITLE.....is memory aid routine description.
PG...............is page location of routine instructions.
BYTS.............is number of bytes Label to RTN inclusive.

FL...............is program file described by first LBL in file.
REG..............is number of registers to copy file.
CALLS............is a list of all ROM Global Labels required to run.

LBL	ROUTINE TITLE	PG	BYTS	FL	REG	CALLS
	ALPHA REGISTER					
AM	Alpha to Memory	42	17	NS	16	NONE
AD	Alpha Delete Last Character	38	47	ML	64	NONE
MA	Memory to Alpha	276	18	NS	16	NONE
NC	Nth Character	340	112	VK	63	SU
SU	Substitute Character	412	102	VK	63	NONE
VA	View Alpha	422	22	LF	59	NONE
	BLOCK OPERATIONS					
BC	Block Clear	50	18	M2	61	NONE
BE	Block Exchange	54	55	M2	61	NONE
BI	Block Increment	56	19	BL	46	NONE
BM	Block Move	62	56	M2	61	NONE
BR	Block Rotate	64	51	M2	61	NONE
BV	Block View	66	59	SR	40	VA
BX	Block Extremes	68	65	M2	61	NONE
B>	Block Statistics	70	25	M2	61	NONE
DR	Delete Record	128	56	M2	61	NONE
IR	Insert Record	230	22	M2	61	BM
M1	Matrix, Interchange Rows	260	55	M2	61	NONE
M2	Matrix, Multiply Row by Constant	266	20	M2	61	NONE
M3	Matrix, Add Multiple of Another Row	268	37	M2	61	NONE
	CURTAIN					
C?	Curtain Finder	72	67	ML	64	NONE
CU	Curtain Up	106	96	VM	60	NONE
CX	Curtain to Abs. Decimal Location in X	120	94	VM	60	C?, CU
HD	Hide Data Registers	182	127	ML	64	ΣC
UD	Uncover Data Registers	438	14	LF	59	NONE
ΣC	ΣREG Curtain Exchange	464	102	ML	64	NONE
	DISPLAY					
DS	Display Set	132	42	VM	60	NONE
DT	Display Test	134	56	ML	64	NONE
RD	Recall Display Mode	374	36	SR	40	NONE
SD	Store Display Mode	400	20	SR	40	SK
	KEY ASSIGNMENTS					
+K	Additional Key Assignment	24	302	MK	61	DC, VA
1K	First Key Assignment	30	312	MK	61	+K, DC, VA
A?	Assignment Register Finder	36	24	LF	59	LF, E?, 2D, OM
CK	Clear Key Assignments	94	42	LF	59	E?, 2D, OM
MK	Make Multiple Key Assignments	278	401	MK	61	VS, LF, E?, 2D, OM, VA
PK	Pack Key Assignment Registers	360	168	LF	59	E?, 2D, OM
RK	Reactivate Key Assignments	378	32	SR	40	NONE
SK	Suspend Key Assignments	406	30	SR	40	NONE
VK	View Key Assignments	446	222	VK	63	NONE
	LOAD BYTES					
B	Store Part of LB	26	66	LB	71	NONE
BL	BLDSPEC Inputs for LB	58	37	BL	46	QR
FL	Flag Inputs for LB	166	62	BL	46	QR
L	Load Part of LB	238	357	LB	71	VA, VS, RT, 2D, PD, QR, OM, XD, DC
LB	Load Bytes	242	449	LB	71	VA
XL	XROM Inputs for LB	460	24	SM	26	QR
	MATHEMATICS					
BD	Base B to Base Decimal	52	58	BD	53	NONE
BX	Block Extremes	68	65	M2	61	NONE
B>	Block Statistics	70	25	M2	61	NONE
CA	Complex Arithmetic	74	38	CA	38	NONE
CJ	Calendar Date to Julian Day Number	86	58	BD	53	NONE
CM	Combinations	96	37	BD	53	NONE
CV	Curve Fit	110	42	CV	42	BC
DF	Decimal to Fraction	124	68	FR	36	NONE
EX	Exponent of X	140	30	VM	60	NONE
FI	Financial Calculations	148	47	FI	47	NONE
FD	First Derivative	144	123	IG	43	SD, SK, RD
FR	Fractions	170	84	FR	36	VA
GN	Gaussian RN Generator	176	35	FR	36	RD
IG	Integrate	220	131	IG	43	NONE
JC	Julian Day Number to Calendar Date	234	98	BD	53	NONE
M1	Matrix, Interchange Rows	260	55	M2	61	NONE
M2	Matrix, Multiply Row by Constant	266	20	M2	61	NONE
M3	Matrix, Add Multiple of Another Row	268	37	M2	61	NONE
M4	Matrix, Register Address to (i,j)	270	20	M2	61	QR
M5	Matrix, (i,j) to Register Address	274	38	M2	61	NONE
MT	Mantissa of X	338	26	VM	60	NONE
	MATHEMATICS CONTINUED					
NP	Next Prime	346	45	FR	36	NONE
PM	Permutations	364	32	BD	53	NONE
PR	Pack Register	368	21	M2	61	UR
QR	Quotient Remainder	372	21	IF	60	NONE
RN	Random Number Generator	380	29	FR	36	NONE
SE	Selection Without Replacement	402	27	SM	26	RN
SV	Solve Routine	416	51	IF	43	NONE
TB	Base Ten to Base B	430	90	BD	53	VA
UR	Unpack Register	440	23	M2	61	NONE
VM	View Mantissa	450	26	VM	60	MT
	MATRIX					
BX	Block Extremes	68	65	M2	61	NONE
B>	Block Statistics	70	25	M2	61	NONE
M1	Matrix, Interchange Rows	260	55	M2	61	NONE
M2	Matrix, Multipley Row by Constant	266	20	M2	61	NONE
M3	Matrix, Add Multiple of Another Row	268	37	M2	61	NONE
M4	Matrix, Register Address to (i, j)	270	20	M2	61	QR
M5	Matrix, (i, j) to Register Address	274	38	M2	61	NONE
	MEMORY					
E?	.END. Finder	136	23	IF	60	2D
EP	Erase Program Memory	138	74	VM	60	PD, 2D, QR, C?, OM, GE, Ab
F?	Free Register Finder	142	26	MK	61	LF, E?, 2D, OM
IP	Initialize Page	228	60	BL	46	NONE
LF	Locate Free Register Block	248	81	LF	59	F?, 2D, OM
ML	Memory Lost Resize to 017	296	36	ML	64	DC, GE, Ab
MS	Memory to Stack	336	31	BL	46	NONE
OM	Open Memory	354	29	IF	60	NONE
PR	Pack Register	368	21	M2	61	UR
PS	Page Switch	370	126	BL	46	TN, DC, XE, GE, Ab, OM, S?, C?, VA
RX	Recall from Absolute Address in X	384	23	IF	60	OM
S?	Size Finder	398	26	ML	64	C?
SM	Stack to Memory	408	36	SM	26	NONE
SX	Store Y in Absolute Address X	424	16	IF	60	OM
UR	Unpack Register	440	23	M2	61	NONE
VS	Verify Size	452	44	VM	60	NONE
Σ?	Sigma REG Finder	462	18	ML	64	C?
	MISCELLANEOUS					
CJ	Calendar Date to Julian Day Number	86	58	BD	53	NONE
GE	Go to .END.	174	84	IF	60	Ab
IF	Invert Flag	216	56	IF	60	NONE
JC	Julian Day Number to Calendar Date	234	98	BD	53	NONE
RF	Reset Flags	376	17	ML	64	NONE
T1	Beep Alternative	428	35	BL	46	NONE
TN	Tone N (0-127)	432	34	VM	60	DC, XE
VF	View Flags	444	101	SM	26	IF, VA
	NON-NORMALIZED NUMBERS					
2D	Decode 2 Bytes to Decimal	34	61	IF	60	NONE
CD	Character to Decimal	84	63	VM	60	NONE
DC	Decimal to Character	122	68	LF	59	NONE
HN	HEX to NNN	184	107	NH	33	NONE
NH	NNN to HEX	342	120	NH	33	NONE
NR	NNN to Recall	350	32	NS	16	NONE
NS	NNN Store	352	25	NS	16	NONE
XD	HEX to Decimal	454	36	LB	71	QR
	PERIPHERALS					
BA	Barcode Analyzer	46	337	BA	49	NONE
CP	Column Print Formating	98	56	LG	28	NONE
HA	High Resolution Histogram with Axis	176	92	LG	28	NONE
HP	High Resolution Plot	188	586	MP	86	IF
HS	High Resolution Histogram	208	92	LG	28	NONE
LG	PPC Logo	252	47	LG	28	NONE
MP	Multiple Variable Plot (1-9)	298	13	MP	86	HP
PO	Paper Out	366	14	NS	16	NONE
XE	XROM Entry	456	58	ML	64	NONE
	PROGRAM POINTER					
Ab	Alpha Store b	44	99	IF	60	NONE
CB	Count Bytes	82	14	IF	60	PD, 2D, QR,
DP	Decimal to Program Pointer	126	30	IF	60	QR, DC,
PA	Program Pointer Advance	356	13	IF	60	PD, QR, DC
PD	Program Pointer to Decimal	358	24	IF	60	2D, QR
Rb	Recall b	386	9	NS	16	NONE
Sb	Store b in ROM	426	8	SR	40	NONE
XE	XROM Entry	456	61	ML	64	NONE
	RETURN STACK					
LR	Lengthen Return Stack	254	40	SR	40	NONE
RT	Return Address to Decimal	382	29	IF	60	2D PD, QR
SR	Shorten Return Stack	410	59	SR	60	Sb
	SORTS					
AL	Alphabetize X & Y	40	105	VK	63	NONE
S1	Stack Sort	388	46	S1	47	NONE
S2	Small Array Sort (≤32)	390	124	S1	47	NONE
S3	Large Array Sort (>32)	394	159	S1	47	NONE

ABSTRACTS

+K - ADDITIONAL KEY ASSIGNMENTS

This routine provides a non-prompting method for making additional key assignments after initial setup work has been completed by either **MK** or **1K**. It is used mainly as a subroutine in programs that set up their own key assignments.

-B STORE PART OF **LB**

This routine allows bytes to be loaded under the control of a users program. **L-** must be executed to initialize this programmable byte loader which loads the decimal byte in the X register.

1K - FIRST KEY ASSIGNMENT

This routine is a subroutine version of **MK** which is non-prompting and may be used in a user program to make a key assignment. **1K** may also be used from the keyboard.

2D - DECODE TWO BYTES TO DECIMAL

This routine decodes the last two bytes of the X register to their decimal equivalents. It is useful for decoding program pointers, which consist of two right-justified bytes.

A? - ASSIGNMENT REGISTER FINDER

This routine finds the number of key assignment registers that are in use, placing this result in the X register.

AD - ALPHA DELETE LAST CHARACTER

This routine removes the rightmost character from the alpha register. It is the equivalent of manually going into ALPHA mode and appending a backarrow.

AL - ALPHABETIZE X & Y

This is a general-purpose alphabetizing subroutine. It compares two alpha strings and, if they are not already in proper order, exchange them. This routine may be used in two different modes. In the direct mode, ALPHA strings in the X and Y registers are alphabetized. In the indirect mode, registers designated by the contents of X and Y are alphabetized.

AM - ALPHA TO MEMORY

The AM routine stores the contents of the Alpha register into a block of data registers defined by a bbb.eeeii formatted control number. The control number is the only required input for **AM** and its inverse routine **MA** .

Ab - ALPHA STORE b

This routine stores the contents of the ALPHA register in register b using ASTO b. This provides an ultra-fast ROM entry capability similar to **XE**, but with no return back to RAM.

BA - BARCODE ANALYZER

This program analyzes single lines of HP41C barcode for barcode type and additional information on its contents. After executing **BA**, the display prompts 'SCAN' and the user scans the line in question. Then the 82143A printer (which is required) prints the barcode type by number and abbreviated name, the value of each 8-bar byte individually in binary, decimal, hex, and equivalent printer ACCHR character if possible, and finally the computed checksum from bytes #2 thru the final byte of 8 bars. This checksum may be compared to the value of byte #1 (the barcode checksum) to evaluate whether the barcode row is valid or will yield an error.

BC - BLOCK CLEAR

This is the block clear routine and is used to store zeros in a block of registers. **BC** uses the complete form of the general block control word bbb.eeeii and can thus be used to clear blocks of consecutive registers or can be used to skip over registers within a block.

BD - BASE B TO BASE DECIMAL

This is a base conversion routine from base b to base 10 where 2<=b<=25. This routine takes advantage of the alpha capabilities when using bases greater than 10. The routine also employs synthetic instructions. The number input is assumed to be in the alpha register when this routine is called. The base b is to have been previously stored. The resulting number in base 10 is left in X. This routine is the inverse of the routine **TB**.

BE - BLOCK EXCHANGE

This routine was inspired by the HP-67/97 Primary-Secondary exchange function. But **BE** is far more versatile as it handles any size blocks anywhere in data memory. Moreover, the blocks need not consist of consecutive registers. **BE** uses the complete form of the general block control word bbb.eeeii. The parameters for the two blocks are completely independent (the blocks may even overlap).

BI - BLOCK INCREMENT

This routine may be used to load a defined block (bbb.eeeii) of registers with zero, a numerical constant, or an incrementing (or decrementing) sequence of numbers when the start and increment values are provided as inputs.

BL - BLDSPEC INPUTS FOR **LB**

BL is used to process seven BLDSPEC numbers to convert them into the equivalent bytes that would be used to represent the BLDSPEC "character" as an alpha text line in a program. **BL** is a supporting program for LB and is intended to be used manually from the keyboard as a programming aid.

BM - BLOCK MOVE

This routine is called block move and applies to any block of consecutive data registers. The routine will move the block anywhere within the defined data register area. Input to BM requires the register number of the first register in the block, the register number which will be the destination of the first register, and finally the number of registers within the block.

BR - BLOCK ROTATE

This routine is called block rotate and applies to any block of consecutive data registers. This routine was inspired by the roll up and roll down functions which apply to the XYZT stack registers. Input to BR is the number of the first register in the block and ±n where n is the number of registers within the block. The sign of n determines the direction of the rotation.

BV - BLOCK VIEW

The BV routine allows rapid viewing (or listing if an 82143A printer is connected and on) of a block of registers defined by the block control number of the format bbb.eeeii. Non-zero data is sequentially displayed along with the appropriate register number. Zero register contents are skipped. A pause is added to the display time if flag 9 is set, a STOP if flag 10 is set.

BX - BLOCK EXTREMA

This routine is called block extrema and may also be considered part of the matrix group since it can be used to determine pivoting operations. This routine will find the largest or smallest element in any block of registers, including any row or column of a matrix. By setting a flag, absolute values of the numbers are used. The actual max/min values as well as their register addresses are returned.

BΣ - BLOCK STATISTICS

This routine is called block statistics, but may be considered part of the matrix group since it is designed to compute vector dot products. Given the appropriate input parameters, this routine can be used to compute matrix products (multiply a row in one matrix by a column in another matrix).

C? - CURTAIN FINDER

This routine provides the absolute address of the "Curtain" which separates data and program memory. This routine is used by S? to find the current SIZE.

CA - COMPLEX ARITHMETIC

This is a complex arithmetic program which employs an infinite complex stack with push and pop operations. Each complex pair must be pushed onto the complex stack from X and Y where pairs are assumed to be in rectangular form. All number operations leave their results on the complex stack as well as in X and Y. The functions provided are: addition, subtraction, multiplication, division, natural log and anti-log, complex Y to the complex X power, complex sine, and complex cosine, a complex X exchange Y function and a complex Last X function. The complex stack may be initialized or cleared at any time, although for the majority of applications this needs to be done only once. The first element pushed onto the complex stack after clearing is considered to be on the top of the stack and may be replicated as many times as desired.

CB - COUNT BYTES

This routine can be used to find the number of bytes in program memory between any two lines by decoding two program pointers supplied by the user (obtained by using a RCL b key assignment).

CD - CHARACTER TO DECIMAL

This routine is a character decoding subroutine that will handle up to 15 characters one by one. Each execution of CD decodes the rightmost character in the alpha register to a decimal number between 0 and 255 (from the byte table). The rightmost character in the alpha register may or may not be removed from the alpha register, under the control of user flag 10.

CJ - CALENDAR DATE TO JULIAN DAY NUMBER

This is a calendar routine which computes the Julian Day Number of a given day. The valid range is from March 1 year 0. This routine can be used to compute the day of the week or the number of days between two dates. Gregorian or Julian calendar dates may be input depending on a flag setting. The input is of the form with the year in Z, the month in Y, and the day in X. This routine is the inverse of the routine JC.

CK - CLEAR KEY ASSIGNMENTS

This routine clears all function key assignments and anything else below the permanent .END.. Global label key assignments are inactivated until a program card is read in.

CM - COMBINATIONS

This routine computes the number of combinations of n objects taken k at a time. This routine selects the optimum input parameters to minimize overflow errors and execution time.

CP - COLUMN PRINT FORMATTING

This routine aligns numeric data into columns for printing tables or lists with the HP82143A printer. A skip index value for each numeric column keeps decimal points in constant position. While CP only adds a single numeric column to the printer's buffer (a single line at a time), it may be called repeatedly to build multiple columns across the 24-character printed line. Columns of ALPHA information may be accumulated at any time by conventional use of the ACA function, to create virtually any combination of multiple numeric and/or ALPHA columns in printed output. A 'printer preparation form' aids the user in planning output before programming, in order to eliminate the trial and error period.

CU - CURTAIN UP

By modifying the contents of the c register, this routine allows the user to raise and lower the curtain which separates the numbered data registers from program memory. This effectively renumbers the data registers and allows a program to hide data from a subroutine.

CV - CURVE FIT

This program will give the curve of best fit to a set of data points. The four standard curve types are: linear (1), exponential (2), logarithmic (3), and power (4) curves. This program will compute the coefficients in the equation of a given curve type as well as give the coefficient of determination, a measure of the goodness of fit. Once a curve type has been selected predictions may be made for either new x or new y values when a y or x value is input. Although logs are used, data input may be negative for either x or y in the case of a linear fit, x may be negative in the exponential fit, and y may be negative in the logarithmic fit. When all data input are positive, a best fit function is provided which will select the best curve type based on the largest coefficient of determination.

CX - CURTAIN TO ABSOLUTE DECIMAL LOCATION IN X

By modifying the contents of the c register, this routine allows the curtain to be moved to an absolute address specifiey by a decimal number in the X register.

DC - DECIMAL TO CHARACTER

This routine is the basic byte building routine used by **MK** and **LB**. It converts a decimal input between 0 and 255 to a byte, which is appended to alpha. This permits arbitrary strings of bytes to be assembled simply by specifying the corresponding decimal codes in sequence.

DF - DECIMAL TO FRACTION

This routine is a decimal to fraction routine. Any decimal value may be input. The output is a fraction whose approximation to the decimal input depends on a previously stored display setting. Setting a flag automatically displays the resulting fraction in the alpha register.

DP - DECIMAL TO PROGRAM POINTER

This routine converts a decimal number in the X register to a RAM pointer usable for a STO b command. **DP** is used in **PA** to move a program pointer.

DR - DELETE RECORD

This routine is called delete record and can be considered part of a data base management system. **DR** applies to files consisting of fixed length records where each record is a block of consecutive data registers. **DR** is a special block move routine which deletes a given record from the file and moves the remaining files into the space occupied by the deleted record so that the data area is used as efficiently as possible. See also the routine **IR**.

DS - DISPLAY SET

This routine provides a capability similar to the HP-67/97 DSP function, which sets the number of decimal places to be displayed without changing the display mode type (FIX, ENG OR SCI).

DT - DISPLAY TEST

This routine allows the user to verify that all of the display segments work. It turns on 12 commas then all of the other display elements.

E? - END FINDER

This routine determines the absolute decimal address of the .END. in program memory by decoding the pointer from status register c.

EP - ERASE PROGRAM MEMORY

This routine provides a means to clear user program memory (RAM) without destroying data, key assignments, and status information.

EX - EXPONENT OF X

This routine isolates the exponent of the number in the X register. It is faster and more accurate that LOG, INT.

F? - FREE REGISTER FINDER

This routine finds the number of unused registers open for program or data use. It provides a programmable equivalent to the manual procedure of switching to PRGM mode at line 00.

FD - FIRST DERIVATIVE

This routine will approximate the first derivative of a function at a point in one of two ways. A quick approximation may be made using a step size that the user inputs, or an adaptive procedure may be used which automatically searches for the optimal step size. A four-point interpolation/approximation is used. In the adaptive routine setting a flag allows the user to view convergence of the optimal step size. The routine may also be used to compute partial derivatives.

FI - FINANCIAL

This is a complete financial program which uses the top two rows of keys to either input or solve for the standard financial values of n i PV PMT FV. This program also handles cases of different compounding and payment periods so that it can be used with Canadian and/or other special types of mortgages. The program also handles the case of continuous compounding. The program thus extends the capabilities of previous HP financial calculators and programs. The program is also highly accurate. The standard financial sign convention is used; money you pay out is negative, money you receive is positive. A Begin/End switch is provided and a status function allows the user to easily determine the state of the toggle functions that the program uses.

FL - FLAG INPUTS FOR **LB**

FL like **BL** and **XL** are programming aid programs intended for keyboard use. The **FL** program takes sequential flags set inputs and produces seven one byte outputs. These are used as inputs for **LB** to produce a synthetic text line that, if stored in the d register, controls all 56 HP-41 flags.

FR - FRACTIONS

This is a program which consists of routines that provide addition, subtraction, multiplication, division, and reduction of fractions. Inputs to all these routines assume the fractions are in the stack and the results are also returned in the stack or may be displayed as fractions in the alpha register by setting a flag.

GE - GO TO .END.

This routine ignores (actually destroys) all pending subroutine returns and instead executes the .END., halting at line 00 of the last program in RAM.

GN - GAUSSIAN RANDOM NUMBER GENERATOR

This is a Gaussian random number generator which yields a Gaussian bell-shaped distribution (also called normal distribution) where the mean and standard deviation are specified by the user. This routine calls the **RN** routine and hence requires a register pointer in X when used. **GN** returns two numbers in the specified range. **GN** also uses the rectangular-polar coordinate conversion functions and should be used in Degrees Mode.

HA - HIGH RESOLUTION HISTOGRAM WITH AXIS

This routine will generate in the print buffer, a high resolution bar-chart bar, extending from a user-prescribed axis position to the input value, for use in charts or histograms. The height of the bar may be from 1 to 168 printer columns, with the axis position in any column. The printed bar always has the axis and the input value as its limits, thus it may extend up or down from the axis, depending upon the position of the input value. The user specifies the fill-character from the standard printer character set, to fill in 7-column portions of the bar. The remaining printer columns are filled by a user-defined fill-column using printer function ACCOL values. Bars created by **HA** are accumulated into the print buffer but not printed, thus allowing additional information to be added later. The inputs to **HA** match those of the printer's REGPLOT routine, allowing either **HA** or REGPLOT to be called for the same inputs.

HD - HIDE DATA REGISTERS

This routine moves the curtain's absolute address up by k registers as specified by a decimal number k in the X register. The former contents of R_{00}-R_{k-1} are then hidden until **UD** is executed to lower the curtain.

HN - HEX TO NNN

This routine converts up to 14 hexadecimal characters in the ALPHA register to a 7-byte non-normalized number in the X register.

HP - HIGH RESOLUTION PLOT

This routine allows between 1 and 9 user functions, written as programs in RAM memory, to be plotted simultaneously in high resolution on the HP82143A printer. High resolution denotes 7 plot points per printed line, allowing each thermal dot position in the X direction to represent a plot point. Plot symbols are individual thermal dots, with different functions identified by different dot sequences on or off in any single print row of 7 plot points. Functions are plotted like the standard PRPLOT function from X and Y limits input by the user. Various options may be implemented, such as changing the standard printed header information, adjusting the order and usage of plot function identifiers and modifying function behavior when values exceed user-specified Y limits.

HS - HIGH RESOLUTION HISTOGRAM

This routine generates in the HP82143A print buffer a high resolution bar-chart bar, whose height extends from the first empty print buffer column position, up to the input value which is scaled between 0 and 1 inclusive. The buffer is not automatically printed, so additional information may be added. The user specifies the fill-character from the standard printer character set, to fill in 7-column sections of the bar. The remaining unfilled columns of the bar are filled by a user-defined fill-column, using printer function ACCOL values. A user-specified plot width between 1 and 168 columns determines the maximum size of any bar.

IF - INVERT FLAG

This routine inverts the state of any of the 56 flags. It can be used to increase the execution speed of programs when the printer is connected to set the low battery indicator, or to do other fun and useful things.

IG - INTEGRATE

This routine uses the Romberg Method of obtaining a numerical approximation for the definite integral of a function. The accuracy depends on the display setting and a flag may be set so the user may view the successive iterations as they converge to the final answer. This routine is similar to the Integrate function on the HP-34C.

IP - INITIALIZE PAGE

This routine stores the last five bytes of register c into the absolute location 256, the last register of a quad memory module. It is used prior to XEQ **PS** to setup a switchable quad for page switching.

IR - INSERT RECORD

This routine is called insert record and is the beginning of a data base management system. **IR** applies to files consisting of fixed length records where each record is a block of consecutive data registers. **IR** is a special block move routine which makes room between two file records for insertion of a new record. See also the routine **DR**.

JC - JULIAN DAY NUMBER TO CALENDAR DATE

This is a calendar routine which computes a calendar date given the Julian Day Number of the date. Input the Julian Day number in X and **JC** returns the date year in Z, the month in Y and the day number of the month in X. Depending on a flag setting, the output date is for the Gregorian or Julian calendar. This routine is the inverse of **CJ**.

L- - LOAD HALF OF **LB**

This routine initializes the programmable byte loader. It is executed by the user's program before executing **-B** . **L-** returns control to the user's program.

LB - LOAD BYTES

This routine loads bytes into program memory from decimal or hexadecimal numbers keyed in by the user.

LF - LOCATE FREE REGISTER BLOCK

This routine provides the location of the highest register used by key assignments and the location of the .END. of program memory. This routine is used by **F?** to find the number of available registers between the .END. and the last key assignment. It is also used by **A?** to find the number of assignment registers used.

LG - PPC LOGO

This routine adds a special-graphics representation of the PPC logo to the current contents of the HP41C print buffer. The logo consists of 21 columns of graphics, filling roughly half of the capacity of the HP82143A buffer. Use the logo to identify material related to PPC, or to members of the club.

LR - LENGTHEN RETURN STACK

Along with the **SR** routine, this routine permits the use of more than six subroutine levels. This routine stores five subroutine return addresses, allowing the user program to call another six subroutine levels. After returning, the user program executes **SR** to extract the stored return addresses from the data registers and add them to the pending subroutine return stack.

M1 - MATRIX 1

This routine will interchange any two rows in a matrix. Input is simply the two row numbers.

M2 - MATRIX 2

This routine will multiply a row in a matrix by a constant. Input is the constant and the row number.

M3 - MATRIX 3

This routine will add a constant multiple of one row to another. Inputs are the two row numbers and the constant.

M4 - MATRIX 4

This routine will give the row and column number of the element in a matrix, given the data register number of the element. This routine is the inverse of **M5** .

M5 - MATRIX 5

This routine will compute the number of the data register for a given element in a matrix, given the row and column numbers for that element. This routine is the inverse of **M4** .

MA - MEMORY TO ALPHA

This routine is the inverse of **AM** and stores into the Alpha register the contents of data registers as defined by the bbb.eeeii formatted control number. **AM** and **MA** provide a convenient means to store and recall the Alpha register for later use.

MK - MAKE MULTIPLE KEY ASSIGNMENTS

This routine extends the capabilities of the ASN function to one-or two-byte codes. This routine enables the user to make synthetic key assignments, ones like STO M, X<>d, byte jumpers, byte maskers, Q-loaders and similar synthetics. This routine makes it much easier for users who do not have access to a cardreader to construct synthetic programs.

ML - MEMORY LOST RESIZE TO 017

This routine provides a SIZE 017 function when used immediately after a MASTER CLEAR. FIX 2 display status is also set.

MP - MULTIPLE VARIABLE PLOT (1-9)

This routine allows between 1 and 9 user functions, written as programs in RAM memory, or multiple sets of numerical values to be plotted simultaneously on the HP82143A printer. Plot symbols are single columns of dots, chosen from printer ACCOL values. Numerical values are plotted like the standard REGPLOT function. User functions are plotted like the standard PRPLOT function from X and Y limits input by the user. Plot resolution is one plotted value per printed line. Many options may be exercised, such as changing standard printed header information, adjusting the order and symbol usage by functions, and changing the behavior when function values exceed user-specified Y limits.

MS - MEMORY TO STACK

MS is the inverse of **SM** . This routine recalls five registers, the lowest numbered register being stored in R06, into the stack in X, Y, Z, T, L order. Any valid register number may be stored in R06, but special considerations must be made for 0, and R02 through R06. See recommendations and warnings in the **MS** section.

MT - MANTISSA OF X

This routine provides the mantissa of the number in the X register, returning the result to the X register.

NC - NTH CHARACTER

This routine is used to pick out a character from the ALPHA register. It replaces a string of up to 24 bytes by one of the ten rightmost bytes. The extracted byte is also stored in the X register.

NH - NNN TO HEXADECIMAL

This routine converts a Non-Normalized Number in the X register to its 14 hexadecimal digit representation in the ALPHA register.

NP - NEXT PRIME

This routine gives the next prime divisor of an integer greater than or equal to a specified trial divisor which may be 2 or any odd number. Pressing R/S automatically gives the next prime divisor. One of the applications of this routine is to find the prime factors of an integer.

NR - NNN RECALL

This routine is used to recall into the X-register an arbitrary seven byte or shorter hexadecimal code string previously stored in a pair of data registers by **NS**

NS - NNN STORE

This routine enables the user to store a Non-Normalized Number or any hexadecimal code string of up to seven bytes into a pair of user selected numbered data registers. The PPC ROM routines **NS** and **NR** provide a means to store Non-Normalized Numbers and recall them without normalization.

OM - OPEN MEMORY

This routine places the curtain's absolute address at 16 (decimal). It is used mainly as a utility subroutine by **MK** and **LB** to permit access to program memory and the key assignment registers.

PA - PROGRAM POINTER ADVANCE

This routine provides a means to move the program pointer relative to its previous location by a selectable increment. This selectable byte jumper is useful for creating synthetic instructions or inspecting postfix bytes of multi-byte instructions.

PD - PROGRAM POINTER TO DECIMAL

This routine decodes a RAM program pointer into the number of bytes from the bottom of program memory. The input to **PD** is normally obtained by using a RCL b key assignment.

PK - PACK KEY ASSIGNMENT REGISTERS

This routine packs the assignment registers left unused by deleted key assignments, freeing them for use in making more key assignments or adding program lines.

PM - PERMUTATIONS

This routine computes the number of permutations of n objects taken k at a time.

PR - PACK REGISTER

This routine is called pack register and complements the **UR** routine. **PR** provides storage of information in a data register in base b encoded form. The uses of this routine were also described in BETTER PROGRAMMING ON THE HP-67/97.

PO - PAPER OUT

This routine consists of five ADV instructions and may be used to move the 82143A printer paper out of the machine or as a 1/5 second delay if the printer is not connected.

PS - PAGE SWITCH

This routine allows the user to switch between initialized pages of RAM memory. It simplifies page switching when using switchable RAMS or a port extender.

QR - QUOTIENT REMAINDER

This routine is a complete MOD function, providing not only y mod x, but also the quotient, (y-y mod x)/x. It is useful for decomposing decimal numbers to alternate base digits and is used by **XD**, converting a two digit hexadecimal number to its decimal representation. It is used by many PPC ROM routines to slice bytes into nybbles.

RD - RECALL DISPLAY MODE

This routine is used to restore the status of flags 16 through 55 of register d after **SD** was used to save them in a data register. It is useful in long programs which destroy the original display mode.

RF - RESET FLAGS

RF sets all flags to their default (MEMORY LOST) status, except for FIX 2. It is useful when a program sets a number of flags and it is desirable to clean up d-register. Another use is to eliminate the speed penalty caused by having the printer connected during program execution.

RK - REACTIVATE KEY ASSIGNMENTS

This routine is used to restore the use of previous key assignments that were temporarily deactivated by the execution of **SK**. This routine reactivates the key assignments that were in effect before **SK** was executed.

RN - RANDOM NUMBER GENERATOR

This routine is a random number generator and can be used to generate uniformly distributed pseudo-random numbers in the range $0 < r < 1$. The resulting random numbers can be re-scaled to produce uniform random numbers within any specified range. Input to this routine requires a register pointer value which points to the register which will hold the starting seed as well as the subsequent random number decimals.

RT - RETURN ADDRESS TO DECIMAL

This routine evaluates the first subroutine return address in RAM when the contents of the b register has been recalled to the X register.

RX - RECALL FROM ABSOLUTE ADDRESS IN X

This routine recalls and normalizes a register located at an absolute address specified in the X register. **RX** is usually used to recall data stored by **SX**.

Rb - RECALL b

This routine performs the synthetic instruction RCL b in the PPC ROM. **Rb** , along with **2D** , provides a simple means to find which port the PPC ROM is plugged into. **Rb** also enables construction of a port-independent PPC ROM entry routine.

S1 - STACK SORT

S1 sorts the stack registers X, Y, Z, and T into an order that has the highest numeric value in X and the lowest in T. The reverse order is obtained if flag 10 is set. Suitable for ordering scores, etc.

S2 - SMALL ARRAY SORT (≤32)

A numerical array as defined by the block control number bbb.eeeii is sorted into an order that has the lowest value in the array stored in bbb. register and the highest value stored in the eee register. **S2** is called as a subroutine by **S3**. Optimum speed is obtained if the array is 32 or fewer registers, but **S2** may be used for larger arrays to take advantage of the ii capability of the control number.

S3 - LARGE ARRAY SORT (>32)

Large arrays of numerical data may be sorted by **S3** using the format bbb.eee to define the registers to be sorted. **S3** is the fastest known numerical sort routine for the HP-41. Any register from R03 up to the limit of the SIZED memory may be sorted.

S? - SIZE FINDER

This routine finds the current SIZE by decoding the absolute address of the curtain found in register c.

SD - STORE DISPLAY MODE

This routine saves flags 16 through 55 in a register defined by X. It is useful at the beginning of programs which alter the display or trig status. **RD** is used to restore these flags.

SE - SELECTION WITHOUT REPLACEMENT

This routine is a selection without replacement routine and can be used to select at random an element from any block of consecutive registers. Subsequent items selected from the block will not be repeated. The data block that this routine selects from can have its data arranged in any order. For example, if this routine is used to deal cards from a deck of cards the cards do not have to be shuffled. In a different light, due to the selection technique used, this routine can be considered as a random shuffler which will scramble data that has been serially ordered. **SE** calls the random number generator routine **RN** .

SK - SUSPEND KEY ASSIGNMENTS

This routine suspends both global label and function key assignments. The key assignments are saved and may be restored by executing **RK** or by reading in a program card. Two registers (lowest number in X) are used to store the key map. This is useful when the local label assignments are needed by a user program, for example **FI** .

SM - STACK TO MEMORY

This routine provides a convenient means to store the five registers X, Y, Z, T, and L into a block of registers as controlled by R06. The lowest of the five registers block is stored in R06. R06 may contain any number, but 0, and 2 through 6 must be used with the considerations described in the **SM** and **SM** write-up.

SR - SHORTEN RETURN STACK

Along with the **LR** routine, this routine permits the use of more than six subroutine levels. The **LR** routine stores five subroutine return addresses, allowing the user program to call another six subroutine levels. After returning, the user program executes **SR** to extract the stored return addresses from the data registers and add them to the pending subroutine return stack.

SU - SUBSTITUTE CHARACTER

This routine is used to edit the alpha register. The ten rightmost bytes in the alpha register may be replaced one character at a time.

SV - SOLVE

This routine is a solve routine which approximates a solution to an equation of the form f(x)=0 using the Secant Method (a simplified form of Newton's Method). Input requires an initial guess and an initial step size. The output will leave the x-value in X which most closely makes f(x)=0. A flag may be set to display the successive approximations as they converge to the final answer. Convergence depends on the initial guess.

SX - STORE Y IN ABSOLUTE ADDRESS X

This routine may be used to store data or program bytes in any desired register in user memory. It permits direct modification of programs and key assignments or storing data in the unused memory space between the .END. and the assignment registers. This is especially useful when "page switching" memory.

Sb - STORE b IN ROM

This routine provides a simple way to transfer program execution to any point in a ROM program from which a RTN is not required.

T1 - BEEP ALTERNATIVE

T1 is the only one of many special sound effect routines that were proposed for the ROM that survived. It is a rapid sequence of synthetic tones that provides a Beep Alternative.

TB - BASE TEN TO BASE B

This is a base conversion routine from base 10 to base b where 2<=b<=19. The base 10 number is assumed to be in the X-register when this routine is called. The resulting number in base b is left in the alpha register and may be automatically viewed by setting a flag. This routine also uses synthetic instructions and is the inverse of the routine **BD** .

TN - TONE N (0-127)

All HP-41 synthetic tones may be heard using **TN**, which may be thought of as an indirect tone instruction. When **TN** is executed, the number in X is converted into a TONE byte and executed. **TN** is a simple example of the power of synthetic programming.

UD - UNCOVER DATA REGISTER

This routine uses the information stored in R00 by **HD** to return the curtain to the position it had prior to the last call to **HD**.

UR - UNPACK REGISTER

This routine is called unpack register and is one of the two data packing routines in the ROM. See also **PR**. This routine is a carry over from the HP-67/97 and allows recalling of information in a data register which has been encoded in base b. The scope of this routine is described in BETTER PROGRAMMING ON THE HP-67/97.

VA - VIEW ALPHA

This routine functions as an alternative to the standard AVIEW. It displays the contents of the alpha register and also prints it if, and only if, the printer is connected, turned on, and enabled, without halting the program execution. **VA** avoids the program halt encountered in a running program when the printer is not connected, flag 21 is set and AVIEW is executed.

VF - VIEW FLAGS

This routine displays the user and system flags which are set. A list of the set flags will be printed if the printer is connected, turned on, and enabled.

VK - VIEW KEY ASSIGNMENTS

This routine displays for each assigned key the assigned key's coordinates on the keyboard. It is similar to the 82143A printer's PRKEYS routine, but doesn't require a printer.

VM - VIEW MANTISSA

This routine allows the user to view the full ten digit mantissa of a number in the X register without changing the display mode and without disturbing the stack.

VS - VERIFY SIZE

This routine prompts the user to resize if the current size is not sufficient as specified by the user's calling program.

XD - HEX TO DECIMAL

This routine converts a two digit hexadecimal number in the alpha register to its decimal representation which is stored in the X register.

XE - XROM ENTRY

This routine provides a means to enter any ROM routine at any point without the need for a global label. This enables the use as a subroutine of a ROM program written for manual use.

XL - XROM INPUTS FOR LB

XL along with **FL** and **BL** are keyboard executed programs that are intended to be programmers aids. **XL** converts two XROM number inputs to the corresponding memory bytes to use as **LB** inputs or MK (and other key assignment programs) inputs. Any XROM instruction may be entered into program memory or assigned to a key, even if the ROM is not available.

Σ? - ΣREG FINDER

This routine provides the number of the first register in the statistical register block. It provides the capability of finding these registers without using the printer's PRFLAGS function.

ΣC - ΣREG CURTAIN EXCHANGE

This routine interchanges the pointers in status register c to R00 and to the statistical block of register, raising the curtain by n registers if preceded by a call to ΣREG n command, then restoring the original curtain the second time it is called.

INTRODUCTION TO SYNTHETIC PROGRAMING

PART I - OVERVIEW

New users of the HP-41C/V, and even many experienced users, may be surprised upon reading the program listings in this manual to encounter a number of 41C program lines that they do not recognize. "STO M" and "RCL b", for example, can not be found in the HP-41C/V owner's manuals; yet they are well defined, quite executable and useful functions. They can be assigned to keys, recorded on cards, etc.--in short, they possess all of the properties of "normal" functions. These new functions are called "synthetic functions", because they are created in the calculator memory by synthesizing together combinations of program bytes that can't be obtained with ordinary keystrokes. A "RCL b" is the result of combining the "RCL" prefix with the "b" postfix (as found normally in "LBL b").

"Synthetic programming" simply refers to any use of synthetic functions in HP-41C programming. Stated most concisely, the synthetic program lines constitute an extension of the normal HP-41 function set. Their usefulness depends on the particular application, and on the programmer's creativity--just like any normal function. If a programmer doesn't have a use for the "LN" function, he doesn't really care whether it's available. But if he needs it, there is no substitute for it. The same applies to synthetic functions. They perform certain operations--if you can use them, they're great; if you can't, you can forget about them.

Historically synthetic instructions have been used in programs ever since the HP-41C was introduced on July 17, 1979. The HP-67 compatibility feature of the HP-41C card reader "translated" certain exponent instructions to produce such program lines as E3, -E3, E-3, etc. These "short form" versions of 1E3, -1E3, and 1E-3 saved a byte and executed faster. From a historical viewpoint HP provided the first synthetic instructions in their Solutions books. After more than two years of using synthetic instructions, it is clear that they are here to stay. George Lithograph has barcode generating programs that are refined to the point of accurately processing all known synthetic instructions. HP's most recent ROM revisions haven't changed the HP-41's processing of these synthetic instructions, and the latest HP-41CV runs synthetic programs just like the earliest machines. Early synthetic programs may have used BUG's that were only found in very early ROM revisions. Synthetic instructions do not require any BUG's to run, except in very rare situations. There is no valid reason to avoid synthetic instructions because we now understand the HP-41C/CV.

The applications of synthetic functions fall into two general categories: program enhancement, and user-machine interaction. For program enhancement, synthetic functions perform certain tasks faster than normal functions, and other tasks that normal functions can't do at all. An example of the latter is the function "RCL d", which recalls a number representing the status of all 56 user and system flags into the X-register. This number can be restored back to its origin at any time via a "STO d" line--thus, the user can control the configuration of all 41C flags with a single program line or keystroke. An example of the second class of application, user-machine interaction, is synthetic key assignments, where multi-keystroke operations like "GTO IND X" can be assigned to a key for single-keystroke execution or program entry.

The application of synthetic programming depends on the user's understanding of two general topics. The first is the structure of HP-41C instructions; the second is the organization of the 41C memory, in particular the nature and roles of the 16 "scratch registers" (so-called by HP), also known as the "status registers". We will consider each of these in turn.

A. HP-41C INSTRUCTIONS

All HP-41C user-generated instructions are coded and stored as strings of binary "bits", i.e., "1's" and "0's" represented electronically. As the Central Processing Unit (CPU) "reads" the binary strings, it translates them into executable processes. This translation is invisible to the user--he presses "+", for example, and sees the result of the addition. To actually perform the additon, however, the CPU had to perform a large number of steps, from decoding which key was pressed, to finding the numbers to be added, and finally to displaying the result. Moreover, it carried out a number of routine "housekeeping" chores, like looking for peripherals that might want attention or checking the battery state.

For the most part, instruction codes are not handled by the CPU as individual binary bits, but are grouped together into groups of 8 bits called "bytes". A byte is the smallest unit of program code over which the user has keyboard control. Many 41C instructions are represented as a single byte, and the remainder as multi-byte groups; none are coded with fewer than 8 bits.

There are 256 possible values for an 8-bit number, from 0 to 255. These may be conveniently organized into a 16 x 16 matrix, which provides a compact, easily understood representation of the 41C instruction set. The matrix, usually called the "Byte Table" or "Hex Table" is shown in the table of this section. Each entry in a horizontal row has the same initial 4 bits; the entries in a vertical column have the last 4 bits in common. Recall that 4 bits can represent the numbers one through sixteen; hence, a group of 4 bits can be represented by a single digit in the hexadecimal number system. (In the "hex" system, the numbers 10 through 15 are represented by the single characters "A" through "F", respectively.) Each 8-bit byte is thus represented by a two-digit hex number, 00 through FF. For a particular entry in the table, its vertical position identifies its first four bits; its horizontal positon shows the remaining four bits. The particular operation that results from each byte depends upon its relation to other bytes in memory; the table shows each byte's possible roles by the various entries in the box for each byte.

HP-41C COMBINED HEX/DECIMAL BYTE TABLE

HP-41C COMBINED HEX/DECIMAL BYTE TABLE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	NULL 00 ⁻ 0 ◆	LBL 00 01 x̄ 1 ×	LBL 01 02 ▒ 2 x̄	LBL 02 03 ▒ 3 ←	LBL 03 04 x̄ 4 α	LBL 04 05 x̄ 5 β	LBL 05 06 Γ 6 Γ	LBL 06 07 ▒ 7 ↓	LBL 07 08 ▒ 8 Δ	LBL 08 09 ▒ 9 σ	LBL 09 10 ▒ 10 ◆	LBL 10 11 ▒ 11 λ	LBL 11 12 µ 12 µ	LBL 12 13 ∡ 13 ∡	LBL 13 14 ▒ 14 τ	LBL 14 15 ▒ 15 Φ	0
1	0 16 ▒ 16 Θ	1 17 ▒ 17 Ω	2 18 ▒ 18 δ	3 19 ▒ 19 Ä	4 20 ▒ 20 ä	5 21 ▒ 21 Ä	6 22 ▒ 22 ä	7 23 ▒ 23 Ö	8 24 ▒ 24 ö	9 25 ▒ 25 Ü	. 26 ▒ 26 ü	EEX 27 ▒ 27 Æ	NEG 28 ▒ 28 œ	GTO ᵀ 29 ∡ 29 ≠	XEQ ᵀ 30 ▒ 30 £	W ᵀ 31 ▒ 31 ▒	1
2	RCL 00 32 32	RCL 01 33 ! 33 !	RCL 02 34 " 34 "	RCL 03 35 # 35 #	RCL 04 36 $ 36 $	RCL 05 37 % 37 %	RCL 06 38 & 38 &	RCL 07 39 ' 39 '	RCL 08 40 (40 (	RCL 09 41) 41)	RCL 10 42 * 42 *	RCL 11 43 + 43 +	RCL 12 44 , 44 ,	RCL 13 45 - 45 −	RCL 14 46 . 46 .	RCL 15 47 / 47 /	2
3	STO 00 48 0 48 0	STO 01 49 1 49 1	STO 02 50 2 50 2	STO 03 51 3 51 3	STO 04 52 4 52 4	STO 05 53 5 53 5	STO 06 54 6 54 6	STO 07 55 7 55 7	STO 08 56 8 56 8	STO 09 57 9 57 9	STO 10 58 : 58 :	STO 11 59 ; 59 ;	STO 12 60 < 60 <	STO 13 61 = 61 =	STO 14 62 > 62 >	STO 15 63 ? 63 ?	3
4	+ 64 @ 64 @	— 65 A 65 A	* 66 B 66 B	/ 67 C 67 C	X<Y? 68 D 68 D	X>Y? 69 E 69 E	X≤Y? 70 F 70 F	Σ+ 71 G 71 G	Σ− 72 H 72 H	HMS+ 73 I 73 I	HMS− 74 J 74 J	MOD 75 K 75 K	% 76 L 76 L	%CH 77 M 77 M	P→R 78 N 78 N	R→P 79 O 79 O	4
5	LN 80 P 80 P	X↑2 81 Q 81 Q	SQRT 82 R 82 R	Y↑X 83 S 83 S	CHS 84 T 84 T	E↑X 85 U 85 U	LOG 86 V 86 V	10↑X 87 W 87 W	E↑X-1 88 X 88 X	SIN 89 Y 89 Y	COS 90 Z 90 Z	TAN 91 [91 [	ASIN 92 \ 92 \	ACOS 93] 93]	ATAN 94 ↑ 94 ↑	→DEC 95 _ 95 _	5
6	1/X 96 ᵀ 96 ᵀ	ABS 97 a 97 a	FACT 98 b 98 b	X≠0? 99 c 99 c	X>0? 100 d 100 d	LN1+X 101 e 101 e	X<0? A ▒ 102 f	X=0? B ▒ 103 g	INT C ▒ 104 h	FRC D ▒ 105 i	D→R E ▒ 106 j	R→D F ▒ 107 k	→HMS G ▒ 108 l	→HR H ▒ 109 m	RND I ▒ 110 n	→OCT J ▒ 111 o	6
7	CLΣ T ▒ 112 p	X<>Y Z ▒ 113 q	PI Y ▒ 114 r	CLST X ▒ 115 s	R↑ L ▒ 116 t	RDN M [▒ 117 u	LASTX N / ▒ 118 v	CLX O] ▒ 119 w	X=Y? P ↑ ▒ 120 x	X≠Y? Q _ ▒ 121 y	SIGN ⊢ ᵀ ▒ 122 z	X≤0? a ▒ 123 π	MEAN b ▒ 124 l	SDEV c ▒ 125 →	AVIEW d Σ 126 Σ	CLD e ⊢ 127 ⊢	7
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	

HP-41C COMBINED HEX/DECIMAL BYTE TABLE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
8	DEG IND 00 128 ◆	RAD IND 01 129 ×	GRAD IND 02 130 x̄	ENTER↑ IND 03 131 ←	STOP IND 04 132 α	RTN IND 05 133 β	BEEP IND 06 134 Γ	CLA IND 07 135 ↓	ASHF IND 08 136 Δ	PSE IND 09 137 σ	CLRG IND 10 138 ◆	AOFF IND 11 139 λ	AON IND 12 140 µ	OFF IND 13 141 ∡	PROMPT IND 14 142 τ	ADV IND 15 143 Φ	8
9	RCL IND 16 144 Θ	STO IND 17 145 Ω	ST+ IND 18 146 δ	ST− IND 19 147 Ä	ST* IND 20 148 ä	ST/ IND 21 149 Ä	ISG IND 22 150 ä	DSE IND 23 151 Ö	VIEW IND 24 152 ö	ΣREG IND 25 153 Ü	ASTO IND 26 154 ü	ARCL IND 27 155 Æ	FIX IND 28 156 œ	SCI IND 29 157 ≠	ENG IND 30 158 £	TONE IND 31 159 ▒	9
A	XR 0-3 IND 32 160	XR 4-7 IND 33 161 !	XR8-11 IND 34 162 "	X12-15 IND 35 163 #	X16-19 IND 36 164 $	X20-23 IND 37 165 %	X24-27 IND 38 166 &	X28-31 IND 39 167 '	SF IND 40 168 (	CF IND 41 169)	FS?C IND 42 170 *	FC?C IND 43 171 +	FS? IND 44 172 ,	FC? IND 45 173 −	GTO/XEQ IND IND 46 174 .	SPARE IND 47 175 /	A
B	SPARE IND 48 176 0	GTO 00 IND 49 177 1	GTO 01 IND 50 178 2	GTO 02 IND 51 179 3	GTO 03 IND 52 180 4	GTO 04 IND 53 181 5	GTO 05 IND 54 182 6	GTO 06 IND 55 183 7	GTO 07 IND 56 184 8	GTO 08 IND 57 185 9	GTO 09 IND 58 186 :	GTO 10 IND 59 187 ;	GTO 11 IND 60 188 <	GTO 12 IND 61 189 =	GTO 13 IND 62 190 >	GTO 14 IND 63 191 ?	B
C	GLOBAL IND 64 192 @	GLOBAL IND 65 193 A	GLOBAL IND 66 194 B	GLOBAL IND 67 195 C	GLOBAL IND 68 196 D	GLOBAL IND 69 197 E	GLOBAL IND 70 198 F	GLOBAL IND 71 199 G	GLOBAL IND 72 200 H	GLOBAL IND 73 201 I	GLOBAL IND 74 202 J	GLOBAL IND 75 203 K	GLOBAL IND 76 204 L	GLOBAL IND 77 205 M	X<>-- IND 78 206 N	LBL -- IND 79 207 O	C
D	GTO -- IND 80 208 P	GTO -- IND 81 209 Q	GTO -- IND 82 210 R	GTO -- IND 83 211 S	GTO -- IND 84 212 T	GTO -- IND 85 213 U	GTO -- IND 86 214 V	GTO -- IND 87 215 W	GTO -- IND 88 216 X	GTO -- IND 89 217 Y	GTO -- IND 90 218 Z	GTO -- IND 91 219 [	GTO -- IND 92 220 \	GTO -- IND 93 221]	GTO -- IND 94 222 ↑	GTO -- IND 95 223 _	D
E	XEQ -- IND 96 224 ᵀ	XEQ -- IND 97 225 a	XEQ -- IND 98 226 b	XEQ -- IND 99 227 c	XEQ -- IND100 228 d	XEQ -- IND101 229 e	XEQ -- IND102 230 f	XEQ -- IND103 231 g	XEQ -- IND104 232 h	XEQ -- IND105 233 i	XEQ -- IND106 234 j	XEQ -- IND107 235 k	XEQ -- IND108 236 l	XEQ -- IND109 237 m	XEQ -- IND110 238 n	XEQ -- IND111 239 o	E
F	TEXT 0 IND T 240 p	TEXT 1 IND Z 241 q	TEXT 2 IND Y 242 r	TEXT 3 IND X 243 s	TEXT 4 IND L 244 t	TEXT 5 INDM[245 u	TEXT 6 IND N/ 246 v	TEXT 7 INDO] 247 w	TEXT 8 IND P↑ 248 x	TEXT 9 INDQ_ 249 y	TEXT10 IND⊢ᵀ 250 z	TEXT11 IND a 251 π	TEXT12 IND b 252 l	TEXT13 IND c 253 →	TEXT14 IND d 254 Σ	TEXT15 IND e 255 ⊢	F
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	

The byte table is organized as much as possible so that bytes in a particular row have a certain amount of similar processing in common. For example, we see that the instructions "STO 00" through "STO 15" occupy the entire row 3. When the processor encounters any of these bytes, it knows that a store instruction is pending, and it can carry out much of the processing before even checking the second 4 bits to find out which user register is addressed.

As mentioned previously, all 41 instructions are coded with one-or-more byte groups. All of the bytes in rows 0 through 8, with the exception of bytes 1D and 1E (1F is not used in this context) represent "one byte functions". That is, when one of these bytes is read to begin a new operation, no additional bytes are required to identify the operation. "72" results in the "SIN" function being executed; "82" results in "GRAD" mode being set. The byte "00" represents the "null" function. This byte is simply a place holder, resulting from program insertions and deletions, and is normally "invisible" in a program display.

The bytes in rows 9 and A, plus bytes CE and CF, may be considered as "prefixes" for two-byte functions. That is, when one of these bytes is encountered, one additional byte must be read to complete the instruction. The second, or "postfix" byte identifies either the user register or the label name (for CF) upon which the operation identified by the prefix byte is to be performed. The postfix role of a byte is shown by the number or single letter immediately below the prefix entry in the corresponding byte table box. Thus we have, for example, 9152 representing "STO 82" (note that 52 in hexadecimal is the same as 82 in decimal), or CF 7C representing "LBL b".

Bytes A0-A7 are reserved for functions found in external peripherals. When an "A" digit starts a prefix, the next bit determines whether the function is found in the 41C (1) or in a peripheral (0).

Since two-byte functions, by definition, occupy more user memory space than one-byte instructions, certain frequently used instructions, LBL 00 through LBL 14, and STO and RCL 00 through 15, are assigned one-byte representations, found in rows 0, 2, and 3 respectively. Additional two-byte functions could have been given similar "shorthand" codes, but that would not have left enough of the 256 codes for the other functions that give the 41C its versatility. We note that the postfix values found in rows 0-7 are duplicated in rows 8-F. The duplication is only apparent; a postfix from the bottom half of the table indicates an indirect operation. While "9120" encodes "STO 32", "91A0" represents "STO IND 32". This rule is modified slightly when the prefix is "AE". In this case, the postfix actually determines the role of the prefix. If the postfix is less than "80" the function is "GTO IND", whereas, a postfix of "80" or greater causes an "XEQ IND". For example, "GTO IND 69" is coded "AE 45", but "XEQ IND 69" is "AE C5".

Rows B and D in the table consist of two- and three-byte GTO's, respectively. In both cases, only the first hex digit is needed to identify the function as a "GTO". The remaining bits of the instructions identify the name of the addressed program (numeric) label and also the location of the label relative to the GTO. The encoding of the label positon is done the first time a program is run, which results in faster execution on subsequent runs, since execution can jump directly to the label without any searches. The encoding, or "compiling", is lost any time the program is edited. The two-byte GTO's have only 7 bits available for location coding, so that the label must be within 112 bytes of the GTO to enable a compilation. The three-byte GTO's have 12 bits for the jump code, so there is no limit to the distance between the GTO and the corresponding label (within the normal range of the 41C memory). Row E contains the (numeric) XEQ's, which are essentially identical to the 3-byte GTO's.

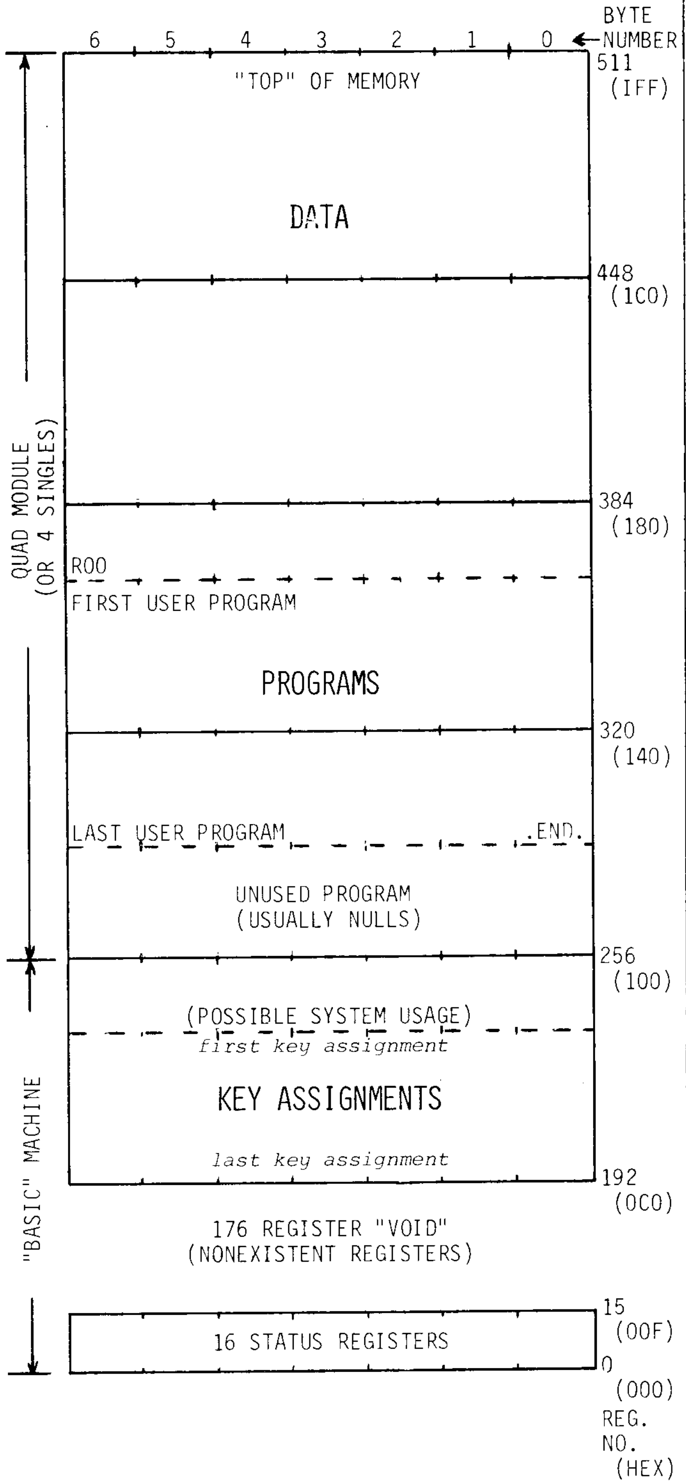
The entries C0-CD initiate "global" instructions including alpha labels and END's, which require three or more bytes. The first hex digit of a global program line identifies it as global, i.e., it is included in the label chain shown in the user catalog (CAT 1). The next 3 digits indicate the distance to the next global line in the chain. The third byte determines whether the code is an alpha label or an END: if the byte is from row F, the line is a label; otherwise it is an END. The portion of alpha labels starting with the third byte, and program text lines are coded nearly the same, starting with a byte from row F. The second digit of the "Fn" byte indicates that the next "n" bytes are to be included in the program line. For text lines, all n bytes are used to encode the key assignment for the label. The text character represented by each byte is shown in the corresponding byte table box. Example: "CAT" is coded as "F3 43 41 54; LBL "CAT" is Ca bc F4 jk 43 41 54", where the digits abc locate the next alpha label in the chain, and the digits jk identify the key (if any) assigned to the label. The only remaining bytes in the table are AF and B0, which have no prefix roles, and 1D and 1E. The latter initiate alpha GTO and alpha XEQ, respectively. These bytes are followed in memory by alpha strings coded as described in the preceding paragraph, which identify the alpha label addressed. There is no compiling of addresses associated with these program lines, since the labels are found by scanning along the global label chain.

B. MEMORY ORGANIZATION

The HP-41C is designed to handle 10-digit decimal numbers. Each decimal digit requires 4 binary bits, i.e., one hex digit; in addition a hex digit each is required to encode the mantissa and exponent signs, and 2 more for the exponent itself. 41C number storage therefore requires 14 digits, or 7 bytes for each number. The user memory (RAM), used both for programs and for data storage, is accordingly organized into 7 byte segments called "registers". Individual registers are identified and located by three digit register numbers. Program memory, including program and data registers, starts at Register 0C0 and continues "upwards" (see Memory Partitioning Map) to a maximum (for the 41CV or a 41C with four memory modules) of 1FF. Individual bytes within a register are numbered from 0 to 6; the full address of a program byte is coded with 4 digits "abcd", where "a" is the byte number and "bcd" is the register number. Single digits in a register are identified by a number 0 thru 13: digits 0 & 1 is byte 0, 12 & 13 is byte 6, etc.

Program execution proceeds (not counting GTO or XEQ jumps) byte-by-byte "downward" through program memory. The 41C has a "program counter" that keeps track of the bytes currently being executed. In the data portion of user memory, each number or alpha string occupies an entire register. For numbers, byte 6 (the first, or leftmost byte of the register) contains the mantissa sign (digit 13, which is: 0 for +, and 9 for -) and the most significant mantissa digit; bytes 5 through 1 store the remainder of the mantissa and the exponent sign (digit 2); byte 0 is the exponent. Alpha strings are coded with byte 6 = "10". Bytes 5 through 0 contain up to six character bytes.

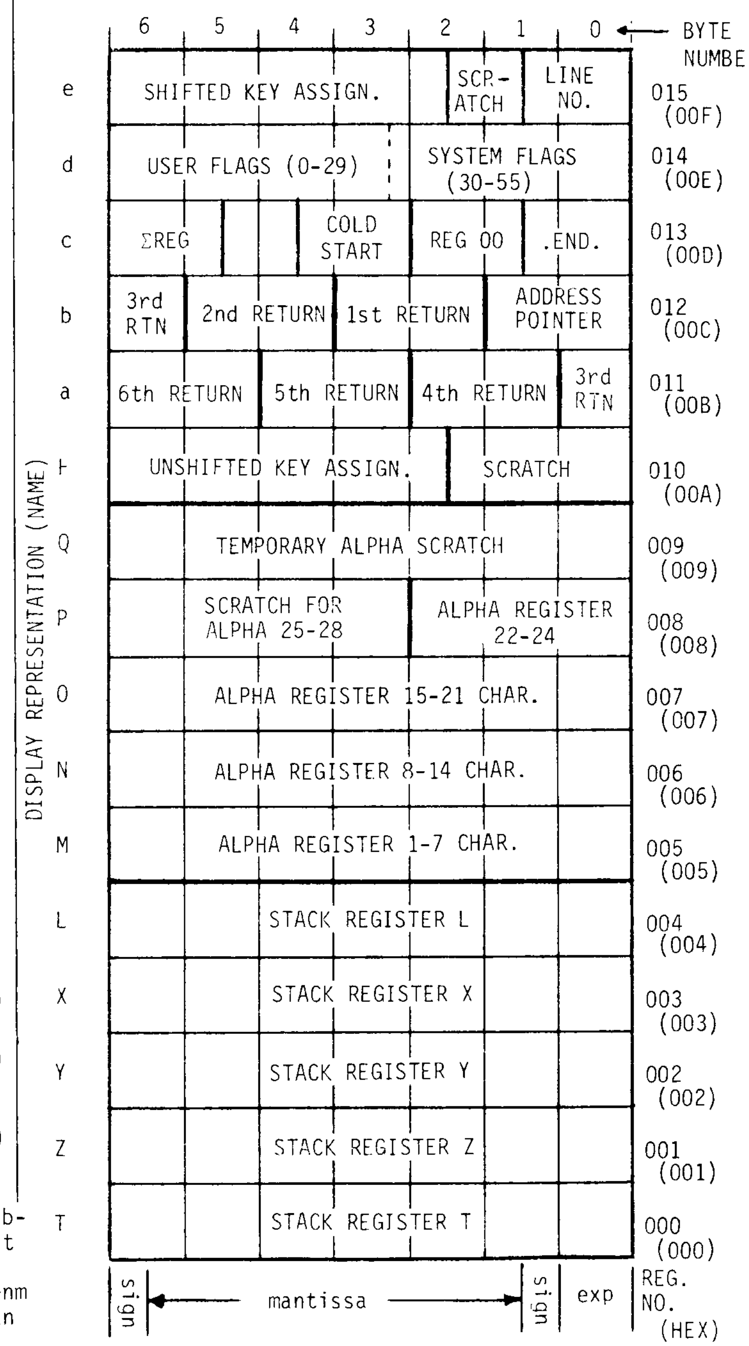
HP-41C/CV RAM MEMORY PARTIONING



The boundary between program and data storage is established simply by storing the address of R00, the first data register, in one of the scratch registers. All data access functions, such as STO nm, RCL nm, or X<>nm access the register whose address is "nm" greater than R00. The "SIZE 1mn" operation moves user memory contents up or down so that the first data register is "1mn" registers below the current top of memory as determined by the number of memory modules installed, and recodes the R00 address accordingly.

User key assignments are coded in program registers starting with register 0C0. Each register can hold two assignments; each new pair of assignments is stored in 0C0, pushing previous assignments upwards in memory. The coding in each assignment register is as follows: Byte 6 is "F0", identifying the register as a key assignment register. Bytes 2 and 1 identify the first assigned function. If the function requires only one byte, then byte 2 is filled with "04". Byte 0 identifies the assigned key, with a value (b-1)a or (b-1)(a+8) corresponding to unshifted or shifted key "ab" (column a, row b), respectively. Bytes 5, 4, and 3 play analogous roles for the second of the two assigned functions.

HP-41C STATUS REGISTER USAGE 0 THRU 15 ABSOLUTE



Between the topmost key assignment register, and the location of the .END. (i.e., the end of user programs) is a space of varying length called the I/O buffer. This space is not used (and may be non-existent if memory is full) by any current 41C peripherals.

At the "bottom" of user memory, occupying locations 000 through 00F (note that there are no registers corresponding to locations 010 through 0BF), is a group of 16 "scratch" registers, also called the "status" registers, since their contents are recorded on track 1 of a status card. Normal user access (i.e., STO, RCL and X<>) is restricted to registers 000, 1, 2, 3, and 4, which are the "stack" registers T, Z, Y, X, and L, respectively. The following figure shows the Status Registers.

The following is a list of the roles of the remaining status registers. Synthetic programming depends heavily on the exploitation of these roles through direct storage and recall of the contents of these registers. Just as register 000, for example, has a symbolic name (register T) arising from the program lines that access it, the status registers take symbolic names from the 41C display of their corresponding (synthetic) store and recall lines.

1. REGISTERS M, N, O & P--THE ALPHA REGISTER

Registers 005 through 008 constitute a block of 28 contiguous bytes that the 41C uses to store alpha strings, i.e., they are "the" alpha register. Normally, only 24 (two display widths) of these characters are considered to be "in" the alpha register; bytes 6, 5, 4, and 3 of register P are occasionally used for other purposes by the 41C.

When an alpha string is entered, the first character is stored in the byte 0 of register M. Each subsequent character also enters byte 0, with previous characters pushed into bytes 1, 2, 3, etc., and eventually into registers N, O, and P as needed.

2. REGISTER Q--ALPHA SCRATCH

Register 009 is used for a number of bookeeping purposes by the 41C. Primary among these is its use as a temporary repository for alpha strings that are not entered into the alpha register, such as are generated when the user keys in an alpha function or label name. It is also used for scratch purposes during key assigning or program text string entry.

3. Register ⊦--THE KEY ASSIGNMENT BIT MAP

When a 41C key is pressed in USER mode, the 41C must check to see whether the key is assigned or in a default state. The first 35 bits of register ⊦ are used for this purpose, one bit to a key. The remainder of the register is used for scratch purposes, typically to store an exponent or a register address during various operations.

4. REGISTERS a AND b--THE PROGRAM COUNTER AND RETURN STACK

The 4-digit program counter mentioned above is stored in bytes 1 and 0 of register 00C. The remaining 5 bytes of 00C (register b), plus all 7 bytes of register 00B (register a), store up to 6 4-digit subroutine addresses. When an XEQ is performed, the program counter value is moved (actually a condensed form, to allow for calls to peripheral routines) to the first return position, bytes 2 and 3 of register b. Each subsequent call pushes the return "stack" to make room for the new program counter; each RTN "drops" the stack so that the first pending return address becomes the new program counter.

5. REGISTER c--MEMORY ORGANIZATION

Three key memory allocation parameters are stored in register 00D (register c). Byte 6 and the last half of byte 5 store the three digit register address of the first sigma register. The address of R00 is encoded in digits 3-5 (byte 2 and half of byte 1); the location of the permanent .END. is stored in digits 0-2. Note that the numerical difference between digits 3-5 and 0-2 is the total number of registers currently allocated to user programs. Digits 6-8 contain the hex number 169, the so-called "cold start constant". The processor checks this value frequently. If it has changed from 169, the calculator does a "cold start"--including clearing of memory. Digits 9 and 10 appear not to be used. They are normally both zero.

6. REGISTER d--THE 56 FLAGS

Each of the 56 bits of register 00E (register d) is one of the system or user flags, starting with the last bit of byte 6 as flag 00 and running through the first bit of byte 0 as flag 55.

7. REGISTER e--MORE KEY ASSIGNMENTS, AND THE LINE NUMBER

The last 35 bits of register 00F (register e) are used as a key assignment bit map for shifted keys, exactly the same as the corresponding bits in register ⊦ are used for unshifted keys. In addition, digits 0, 1, and 2 store the current program line number. During a running program this value is set to FFF. After a program is run, the next time PRGM mode is activated, a new line number is computed by counting down from the preceding END.

C. SYNTHETIC PROGRAMMING

A "synthetic instruction" is any combination of HP-41C bytes that can not be entered into a program or manually executed using normal keystrokes. Various techniques have been developed that enable the construction of arbitrary byte sequences in program memory. When the 41C processor encounters a byte sequence, it must process it, whether it is a normal combination or not. The results of executing non-standard byte combinations often turn out to have useful, practical applications.

There are two generally applicable methods of generating synthetic instructions represented in the PPC ROM. The first, the **LB** program, is used for creation of all non-standard program lines. Any sequence of program bytes can be entered into program memory using this program. The second fundamental program, **MK** , enables the assignment of arbitrary two-byte functions to user keys. Functions so assigned can be manually executed or entered into program merely by pressing a key. Included here are any two-byte peripheral functions, whether or not the peripheral is installed, and any two-byte mainframe function, such as GTO IND X or X<>99.

There is a considerable variety of additional methods that can be used to generate synthetic instructions. For descriptions of the evolution, theory, and application of these techniques, the reader is referred to the references. For our purposes here, LB and MK give the user sufficient control.

Synthetic instructions can be divided into two general classes for discussion: synthetic functions, and synthetic text. We shall consider each separately.

1. SYNTHETIC FUNCTIONS

Of all the possible non-text byte combinations possible, the most important are those that allow direct user access to and control of status registers 005 through 00F. The ability to code and execute functions such as STO b, RCL M, and X<>c is the foundation of all synthetic programming--without these functions, programs such as LB and MK would be impossible.

The existence of the status register access functions derives from the fact that the status registers are RAM registers similar to the rest of user memory registers. Access to registers X, Y, Z, T, and L is a standard feature; access to the remaining status registers was simply not implemented in the 41C. However, using synthetic programming, we can join any postfixes from row 7 to STO, RCL, and X<> prefixes. The happy result is that the resulting instructions execute the same as the normal stack functions. The codes 90 75 through 90 7F, for example, recall the contents of registers 005 through 00F. The display of these instructions looks just like a stack recall function: 90 75 is "RCL M" (RCL | on the printer). The "M" is just an accidental consequence of the operating system. Similarly, we obtain the following table:

BYTE	DISP	PRNT
75	M	|
76	N	\
77	O	]
78	P	↑
79	Q	_
7A	⊢	⊤
7B	a	a
7C	b	b
7D	c	c
7E	d	d
7F	e	e

Let us consider the function "X<>d" as a prototype synthetic function. When executed, it does exactly what its functional form suggests: it exchanges the contents of register X with those of register d. The storage of the original X value into d affects the status of all 56 system and user flags simultaneously--an obvious consequence is the gained ability to set or clear normally inaccessible system flags.

The central point is this: with the existence of synthetic functions, the status registers 005-00F obtain an accessibility equivalent to that of the stack registers. For the alpha registers 005 - 008 (M - P), this accessibility adds a new dimension to user control of alpha strings. For the remaining registers, 009 - 00F, we must remember that these registers play an active role in the organization of 41C execution. The ability to store user-chosen quantities into these registers gives the user enhanced control over the system that is simply not available using the normal function set. Again, a prime example is the X<>d instruction.

One important feature that all status registers have in common is that recall functions (including RCL, X<>, and VIEW) operating on these registers do not result in their contents being normalized. If these operations are applied to the numbered data registers, their contents are normalized, i.e., non-standard hex digit combinations are eliminated.

As an example of the more exotic applications of synthetic functions, consider the following sequence:

```
01  X<>c
02  X<>Y
03  STO 00
04  X<>Y
05  STO c
```

This sequence is the heart of all byte-loading and synthetic key assignment programs, for it allows direct storage into any 41C RAM register, regardless of whether it is in data memory, program memory, or key assignment memory.

In line 01, we presume that the quantity in X, which is exchanged with the current contents of register c, contains the address of the register into which we wish to store as digits 3, 4, and 5.

Following execution of line 01, that register will be designated as R00. Note that as we stored into c, we preserve its original contents by using X<>c rather than STO c. In line 02, we exchange X and Y, then store the original contents of Y into the "new" R00 in line 03. Lines 04 and 05 restore the original contents of c, so that when execution halts, the address of the .END. is restored, that the sigma-register and R00 addresses are valid, and that the cold-start constant is intact. Failure to satisfy any of these conditions will result in immediate MEMORY LOST!

2. SYNTHETIC TEXT LINES

The second general class of synthetic program lines is "synthetic text lines". A synthetic text line is any text line alpha label, GTO or XEQ, that contains any bytes other than the standard display character set. We notice in the byte table that there are a number of characters (some quite useful) that do not appear on the alpha keyboard. In addition, all of the characters in the lower half of the byte table, and many in the upper half, have no special display character assigned, and default to the "starburst" (or "boxed star") character.

The application of synthetic text lines is not limited to their obvious use for placing the full display character set at the user's disposal. The full 128-character printer character set also becomes accessible through program text lines, without the necessity for use of ACCHR, ACSPEC, BLDSPEC, or flag 13, which leads to great savings of program bytes.

Furthermore, a 7-character text line followed by a RCL M places the text bytes into the X-register, where they can be used for a variety of purposes, including storage into other status registers (like the flag register). For arbitrary control of the byte sequence in X, we must utilize general character sequences--synthetic text lines.

D. CONCLUSION

It would take more space than we have here to give a complete list of the practical applications of synthetic programming. As a matter of fact, the programs in the PPC ROM contain the quintessence of synthetic programming--they are the best examples that can be provided. In closing, it should be reemphasized that there is nothing magic about synthetic programming, despite its strange history and "exotic" creation techniques. Synthetic instructions are just instructions that didn't happen to find their way into the Owner's Manual. Their use should be embraced by all serious HP-41C/V programmers.

E. REFERENCES

Essential information on synthetic programming (status registers, program instruction structure, memory structure, etc.) can be found in the following *PPC CALCULATOR JOURNAL* articles

V6N4P11b HP-41C Main Function Table
V6N5P20b HP-41C Postfix Table
V6N6P19d HP-41C Data and Program Structure
V6N8P27 Through the HP-41C with Gun and Camera, by W. C. Wickes (3735).

An extensive step-by-step introduction to synthetic programming is contained in *SYNTHETIC PROGRAMMING ON THE HP-41C* by W. C. Wickes (3735) (see the appendix D on references and accessories).

An introduction to the use of the byte table can be found in this part entitled "HP-41C Combined Hex/Decimal Byte Table".

PART II - HP-41C COMBINED HEX/DECIMAL BYTE TABLE

The Plastic Shirt Pocket Card and How To Use It

The pocket hex table puts an end to the frustration of trying to find a copy of the 8½" x 11" version. It is small enough (68mm x 117mm) to fit easily alongside the HP-41 in its carrying case, so you'll always be ready to do synthetic programming. The pocket hex table will also fit in a standard HP magnetic card holder or in your shirt pocket. Its credit card construction--plastic between plastic--makes it waterproof and durable enough for heavy use. Each table weighs 0.22 ounce.

Despite its small size, the pocket hex table is quite legible. The primary function appears at the top of each of the 256 boxes, while the postfix equivalent appears directly beneath. This arrangement is especially convenient when using **LB**. The decimal equivalent appears at the lower left of each box. At the lower right of each box is the printer character which results when PRA is executed with that byte in the alpha register. In the first half of the hex table (rows 0 through 7) the display characters appear at the center right of each box. Since all display characters from the second half (rows 8 through F) of the hex table are starbursts, these are not shown explicitly. This allows the full indirect postfixes to be shown on the second half of the table.

The pocket hex table also includes some features not seen on previous tables. The display characters are photographically reduced copies of actual display segments, rather than line drawings. Shading is used to denote printer control characters (PPC CJ, V7N6P20) in rows A through E. Accentuated lines at 4-box intervals make it easier to locate a particular byte. Binary equivalents are given along the bottom of each half of the table for convenient binary conversions.

One pocket hex table is supplied with each PPC ROM. The pocket hex table has two minor errors. The display character J is missing the vertical segment at the left, and the postfixes N and IND N show / rather than \ as the printer equivalent. Extras or replacements are available at some dealers who carry Bill Wickes' (3735) Synthetic Programming book. They are also available by mail. (See Appendix D).

A combined hex/decimal byte table (hex table for short) is essential for synthetic programming on the HP-41.

Even with the advanced features of the PPC ROM program **LB**, the hex table is needed to determine the decimal or hexadecimal numeric equivalents of the synthetic instructions you wish to create.

Synthetic instructions are those which cannot be entered directly from the keyboard. For instance E-3 works as well in a program as 1E-3, it's faster, and it saves a byte to boot. But the HP-41 insists on adding the 1 when you press E-3 in PRGM mode. Example 3 below will show you how to permanently remove that superfluous 1. Other examples will show you how to put nonstandard characters into text lines and how to create the powerful status register access instructions.

As noted in Appendix D of the HP-41 Owner's Handbook, many program instructions are made up of a number of pieces. Each piece is an 8-bit code, or byte. For example, the instruction FS? IND 03 begins with an FS? prefix byte. This byte tells the processor that the program line is incomplete, and to interpret the following byte as a postfix in order to complete the instruction. The second byte of the FS? IND 03 instruction appears as IND 03. In this case the same byte would be ENTER↑ if it were not preceded by the FS? byte. The presence of the FS? prefix byte caused the following byte to lose its identity as an ENTER↑ instruction and become the postfix IND 03. A major theme in the examples to follow will be the placement of synthetic postfixes after standard prefixes to create synthetic instructions which can allow otherwise impossible things to be done on your HP-41.

A byte can range in value from 00000000_2 to 11111111_2; however, it is more convenient to use hexadecimal (00_{16} to FF_{16}) or decimal (0_{10}-255_{10}) to express the value of a byte. The Byte Table is based on the hexadecimal representation rc_{16}, where r is the row number (0 through F) and c is the column number. Each of the 256 boxes of the Byte Table shows several possible interpretations for a given byte.

For example, consider the byte $7E_{16}$. The box at row 7, column E looks like this:

AVIEW	
d	Σ
126	Σ

At the top of the box is the primary (prefix) interpretation AVIEW. At the middle left is the postfix interpretation d, as in LBL d. At the bottom left is the decimal equivalent, hex 7E = decimal 126. The middle right entry is the display character produced when this byte appears in the alpha register. The lower right entry is the printer character produced when this byte is printed from the alpha register. (No display characters are shown from rows 8-F since they are all starbursts--14 segments lit. Printer characters from rows 8-F are invisible in program listings, while the shaded characters from rows A-E cause additional unusual behavior when listed.)

The fastest known method for creating synthetic instructions requires a "prefix masker" key assignment. This assignment is made as follows *(Do this precisely as shown):

1. MASTER CLEAR (to MEMORY LOST status)
   This is done by holding down the backarrow key while turning on the calculator, then releasing the key.
2. ASN "+" to the LN key
3. ASN "DEL" to the LOG key
4. Switch to USER mode
5. In PRGM mode do these steps:

```
LBL "T"
CAT 1, R/S immediately with LBL ⊤ T showing
DEL 001  (press LOG, Σ+)
BST      (this takes a while to execute;
          be patient)
GTO .005 (use LN for 005), see LBL 03
DEL 003  (press LOG, √x ), see STO 01
"?AAAAAA", see "?A-----"
```

6. Switch out of PRGM mode.

*Using the PPC ROM routines 1K, +K, or MK, decimal inputs for the "LN" key would be: 247, ENTER↑, 63, ENTER↑, 15. The bootstrapping procedure was discovered by Keith Kendall (5425).

The prefix masker is now assigned to the LN key. Whenever you see the mnemonic PM in the following discussion, press the LN key in USER mode. If you preview this key assignment you'll see XROM 28, 63. When the prefix masker is inserted between two lines of a program it absorbs the first byte of the second (i.e., the following) program line (although this sometimes requires PACKing first). If the absorbed byte was a prefix for a multi-byte instruction then the following (postfix) bytes are free to become "stand alone" functions or prefixes to new functions.

CAUTION--Do not PM at the step immediately preceding an END. That messes up CAT 1 and makes it difficult to BST. If your keyboard ever "locks up", simply remove the battery pack (and the printer if present) for a couple of seconds and replace the printer (if present) for a couple of seconds and replace it. You may still need to MASTER CLEAR to restore operation.

Now let's make some synthetic instructions.

Example 1: Synthesizing X<>M.

```
GTO.. and key in
    01 ENTER↑
    02 X<> IND 06
```

Then GTO .001 and PM (press the LN key). You'll see ⊤-?----⊠. The starburst is the X<> prefix. SST to see

```
03 BEEP
```

This is the IND 06 postfix from X<> , now free to assume its own identity as an instruction. Row 8, column 6 of the byte table shows that line 03 is a hexadecimal 86 byte. This byte is a BEEP instruction, but it becomes IND 06 when preceded by a prefix byte that requires a postfix byte. X<> (hex CE) is such a prefix byte. Now let's edit the exposed postfix byte and reattach it to the X<> prefix. Backarrow line 03 and replace it by RDN. GTO.001, PM, DEL 002, and SST. We have masked the prefix masker, freeing the X<> prefix (the starburst you saw before) from the text line. The exposed X<> prefix immediately absorbed the RDN byte (hex 75) to become X<> M. This synthetic instruction accesses the rightmost seven characters of the alpha register, but in a numerical form. M can be used as an all-purpose scratch register as long as ALPHA is not needed. This synthetic instruction can be freely recorded, packed, or deleted, just like any other instruction.

Example 2: Synthesizing arbitrary text lines.

The procedure of Example 1 can be generalized to allow editing of text characters. This example illustrates the creation of the partially synthetic text line "WE'RE #1".

```
Key in          01 ENTER↑
                02 "WEXRE X1"
```

Then GTO .001 and PM. SST to line 05. You will see

```
05 E↑X-1
```

This is the instruction-equivalent of the character X (see row 5, column 8 of the byte table). Backarrow this line and replace it by RCL 07, which is hexadecimal 27, the instruction-equivalent of the apostrophe character. SST to line 09, backarrow it, and replace it by RCL 03 (hex 23), which will become the # character. Now we need only to re-expose the hidden text prefix (hex F8) to convert these instructions back to characters. GTO .001, PM, DEL 002, and SST to see the result.

Example 3: Synthesizing short form exponents.

Now we are ready to create a synthetic exponent entry line.

```
Key in          01 ENTER↑
                02 1E-3
```

Now PACK (do not GTO ..), GTO .001, and PM. The starburst character is the absorbed 1. Backarrow and SST to see

```
                02 E-3
```

An alternate procedure that does not require packing is to key in lines 01 and 02 as above, GTO .001, insert X<>Y and PM. Then backarrow twice and SST. The X<>Y is a single-byte instruction that extends the range of the prefix masker past the invisible null that precedes all unpacked digit entries.

Example 4: Faster two byte instructions.

Synthetic two-byte instructions can be created faster than was shown in Example 1 by using an alternate method. For instance, to create RCL d key in

```
01 ENTER↑
02 STO IND 16
03 AVIEW
```

The IND 16 postfix is to become a RCL prefix (both hex 90). The AVIEW prefix is to become a postfix d (both hex 7E). GTO .001, PM, backarrow, and SST to see

```
02 RCL d
```

The prefix masker absorbed the STO prefix, releasing the IND 16 postfix. The IND 16 postfix became a RCL prefix, absorbing, in turn, the AVIEW instruction. RCL d recalls all 56 flags to X as a 56-bit number.

This same method can be used to create any two-byte instruction. For example use IND 78 for X<> , IND 17 for STO, IND 22 for ISG. Synthetic postfix instruction-equivalents can be found on row 7 of the byte table.

Example 5: Synthesizing TONE 89.

Create a synthetic TONE. Key in

01 ENTER↑
02 RCL IND 31
03 SIN

GTO .001, PM, backarrow, and SST to see

02 TONE 9

This is actually a synthetic tone, TONE 89, as you'll hear if you SST in RUN mode.

+K - ADDITIONAL KEY ASSIGNMENTS

+K

+K provides a non-prompting facility for making additional key assignments after register checking and other set-up work has been completed by either MK or 1K. It bypasses those portions of 1K which are concerned with set-up and proceeds immediately to make the assignment. +K is primarily used as a subroutine following 1K, but it can be used by itself if flag 20 is clear and flag 07 is set (this triggers the full set-up procedure and suppresses prompting). It can also be called from the keyboard following an MK session, provided data registers 09-11 are intact. For examples of +K's use as a subroutine, see Application Program 1 for 1K, as well as the Application Program listed here.

Example 1: The following program segment assigns VER to the LN key and SF 14 to the e^x key.

167	XROM 1K	-15
133	168	XROM +K
15	14	

BACKGROUND FOR +K

See MK.

COMPLETE INSTRUCTIONS FOR +K

1K allows you to make key assignments under program control.

1. Make sure (or have your program make sure) that the size is at least 12.
2. If you want your control program to automatically make one key assignment, have it place the prefix, postfix, and keycode in Z, Y, and X respectively, and then have it call 1K as a subroutine (XEQ 1K). The key assignment will be made and control returned to your program. (The display of the number of free registers is bypassed.) If an error occurs, then the program will stop with appropriate error message *unless* you have set Flag 25 before calling 1K, in which case an error will cause Flag 25 to be cleared and control returned to your program without the assignment being made (similarly to the machine use of Flag 25), but with the error message in the alpha register in case you want to know what kind of error it was.
3. To make several key assignments under program control, the first assignment can be made by XEQ 1K and each additional assignment made by XEQ +K). (Each routine is called, of course, after placing the appropriate data in Z, Y, and X.) Routine +K does the same as 1K except that it bypasses the register counting (if Flag 20 is set), enabling additonal key assignments to be made much faster. However, if Flag 20 is clear, then +K does include the register count and is then identical to 1K. (Flag 20 is set, by the way, after a register count.) If your program is such that it is not convenient to call 1K for the first assignment and +K for the rest (e.g. involving a loop where the subroutine call appears only once in the program), the +K may be used for *all* of the key assignments as long as you include the two instructions SF 07, CF 20 before calling +K the first time. (SF 07 selects the non-prompting mode.)

Routine Listing For: +K

```
08♦LBL 01
09 XROM "LF"
10 STO 09
11  E
12 +
13 X<>Y
14 STO 10
15 ASTO 11
16 DSE Y
17 GTO 07
18 SF 20
19 FC?C 09
20 GTO 13

21♦LBL 02
22 RCL 09
23 XEQ 11
24 "REG FREE: "
25 RCL d
26 FIX 1
27 ARCL Y
28 STO d
29 XROM "VA"
30 TONE 6
31 PSE

32♦LBL 03
33 "PRE↑POST↑KEY"
34 CLST
35 XROM "VA"
36 TONE 7
37 STOP
38 GTO 14
   LIST 152

42♦LBL "+K"
43♦LBL 14
44 STO 08
45 RDN
46 STO 07
47 RDN
48 STO 06
49 CF 09
50 RCL 10
51 SIGN
52 FS? 20
53 X≠0?
54 GTO 01

55♦LBL 13
56 RCL 08
57 INT
58 X=0?
59 FS? 07
60 FC?C 20
61 GTO 02
62 X<0?
63 SF 09
64 ABS
65 STO Z
66 44
67 -
68 ABS
69 2
70 X<Y?
71 DSE T
72 R↑
73 STO Y
74  E1
75 ST/ Z
76 MOD
77 8
78 *
79 ENTER↑
80 CF 08
81 LASTX
82 FS? 09
83 ST+ Y
84 R↑
85 INT
86 X≠0?
87 X>Y?
88 GTO 08
89 R↑
90 +
91 ST+ Z
92 X<>Y
93 X<=Y?
94 CLX
95 X≠0?
96 SF 08
97 +
98 36
99 -
100 X>0?
101 GTO 08
102 FC? 09
103 RCL ⊤
104 FS? 09
105 RCL e
106 FC? 08
107 GTO 14
108 STO [
109 "⊢*"
110 X<> [

111♦LBL 14
112 X<> d
113 FS? IND Y
114 GTO 09
115 SF IND Y
116 X<> d
117 FC? 08
118 GTO 14
119 STO [
120 ARCL 10
121 X<> \

122♦LBL 14
123 FC? 09
124 STO ⊤
125 FS?C 09
126 STO e
127 "**"
128 FS? 10
129 ARCL 11
130 X<> Z
131 RCL 07
132 RCL 06
133 XROM "DC"
134 XROM "DC"
135 XROM "DC"
136 FS?C 10
137 GTO 14
138 "⊢♦♦♦"
139 ASTO 11
140 SF 10

141♦LBL 14
142 RCL 09
143 RCL 10
144 X<> c
145 RCL [
146 STO IND Z
147 X<>Y
148 X<> c
149 CLST
150 CLA
151 FC? 10
152 ISG 09
153 SF 20
154 FS? 07
155 RTN
156 FS? 20
157 GTO 03
158 "DONE, NO MORE"
159 SF 09
160 GTO 14

161♦LBL 07
162 "NO ROOM"
163 CF 20
164 CLST

165♦LBL 14
166 FS?C 25
167 RTN
168 XROM "VA"
169 TONE 7
170 TONE 3
171 STOP
172 GTO 01

173♦LBL 08
174 "NO SUCH KEY"
175 GTO 14

176♦LBL 09
177 X<> d
178 "KEY TAKEN"

179♦LBL 14
180 CF 09
181 CF 20
182 FS?C 25
183 RTN
184 XROM "VA"
185 TONE 3
186 PSE
187 "KEYCODE?"
188 CLST
189 RCL 08
190 XROM "VA"
191 TONE 7
192 STOP
193 STO 08
194 GTO 01
   LIST 013

197♦LBL 11
198 INT
199 LASTX
200 FRC
201  E3
202 *
203 X<>Y
204 .5
205 FC? 10
206 SIGN
207 -
208 -
209 END
```

TECHNICAL DETAILS

XROM: 10,03	+K	SIZE: 012

Stack Usage: ALL CLEARED
- 0 T: USED
- 1 Z: PREFIX INPUT
- 2 Y: POSTFIX INPUT
- 3 X: KEYCODE INPUT
- 4 L: USED

Alpha Register Usage: ALL CLEARED
- 5 M:
- 6 N:
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: ALTERED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED BUT RESTORED
- 14 d: USED BUT RESTORED
- 15 e: ALTERED

ΣREG: UNCHANGED

Data Registers:
- R00: ONLY REGISTERS 6-11 ARE USED
- R06: PREFIX
- R07: POSTFIX
- R08: KEYCODE
- R09: INDEX FOR STORAGE
- R10: Rc FOR LOWERED CURTAIN
- R11: FIRST ASSIGNMENT OF PAIR

Flag Usage:
- 04: NOT USED
- 05: NOT USED
- 06: NOT USED
- 07: PROMPTLESS VERSION
- 08: USED
- 09: USED
- 10: USED
- 20: REGISTER CHECK DONE
- 25: CLEARED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 3

Global Labels Called:

Direct	Secondary
LF	E?
VA (IF FLAG 7 CLEARED)	OM
DC	2D
PART OF MK	PART OF GE

Local Labels In This Routine: 07, 08, 09, 11, 13, 14

Execution Time: FOR FLAG 20 SET: 5½ SEC.; FLAG 20 CLEAR: 8.6 - 13.5 SEC.

Peripherals Required: NONE

Interruptible?	NO
Execute Anytime?	NO
Program File:	MK
Bytes In RAM:	302
Registers To Copy:	61

Other Comments:

4. Routines 1K and +K can also be executed from the keyboard, after placing the required inputs in the stack. When routine +K is executed, it will continue key assignments using the prompting version if MK was previously executed (Flag 07 clear), or the non-prompting version if 1K was previously executed (Flag 07 set).

WARNING: Don't use +K by itself (without either MK or 1K having been executed) unless you are sure that Flag 20 is clear, and you have checked Flag 07 to make sure you are getting the version you want. The same warning (to make sure Flag 20 is cleared) applies if the size has been changed, key assignments have been manually added or deleted, key assignments have been packed, or R_{09}-R_{11} have been altered.

APPLICATION PROGRAM 1 FOR +K

The following program AROW (assign row) may be useful for experimenting with one-byte synthetic key assignments. It prompts for the row number (0 to 15) and then makes the 16 key assignments with prefix 00 and postfix in that row. That is, if row M is selected, it makes the assignments whose hex bytes are 00 MN where N = 0, 1, ..., E, F. The assignments are made to the lower 4 x 4 array of shifted keys, in the order -51 to -59, -61 to -64, -71 to -74 -81 to -84. If any of these is already occupied the program will stop for you to clear the key. If desired, XROM CK could be inserted after line 04 to clear all assignments first. Also, lines 02-03 and 30-31 can be eliminated if this program is to be called as a subroutine elsewhere. Further modifications could allow assigning of the form jk MN where j, k, and M are constant and n = 0, ..., F, or varying the prefix byte over a row of the hex table. Time for the present version is about 1½ minutes.

```
01♦LBL "AROW"      18♦LBL 01
02 "ROW?"          19 CLST
03 PROMPT          20 RCL 07
04 16              21 RCL 05
05 *               22 CHS
06 ENTER↑          23 XROM "+K"
07 CF 25           24 ISG 05
08 15              25 GTO 14
09 +               26 6.01
10 E3              27 ST+ 05
11 /               28♦LBL 14
12 +               29 ISG 07
13 STO 07          30 GTO 01
14 51.054          31 CLST
15 STO 05          32 BEEP
16 CF 20           33 END
17 SF 07
```

LINE BY LINE ANALYSIS OF +K See MK.

CONTRIBUTORS HISTORY FOR +K

The programmable key assignment capabilities of the PPC ROM, consisting of the 1K and +K programs, were conceived and implemented by Roger Hill (4940).

FURTHER ASSISTANCE ON +K

Call Roger Hill (4940) at (618) 656-8825.
Call Keith Jarett (4360) at (213) 374-2583.

-B - STORE PART OF LB

L- and -B together comprise a subroutine version of LB . L- initializes the byte loading process without any prompting, returning to the calling program. -B is then used to load each byte from its decimal equivalent under program control.

Example 1: The following program segment prompts for input and loads an XROM instruction into program memory (after the user has supplied the usual LBL "++" ++...+ XROM LB sequence). This program checks for sufficient SIZE, converts the XROM numbers Y and X to decimal codes for LB , and loads two bytes from the decimal codes. It then prompts for another pair of XROM numbers.

```
01 LBL "XLB"          13 XROM XL
02 12                 14 X<>Y
03 XROM VS            15 STO 05
04 FC?C 25            16 X<>Y
05 PROMPT             17 XROM -B
06 XROM L-            18 RCL 05
07 CF 22              19 XROM -B
08 LBL 00             20 GTO 00
09 "XROM Y, X  ?"     21 LBL 01
10 PROMPT             22 CF 09
11 FC?C 22            23 XROM -B
12 GTO 01             24 END
```

To use "XLB" first key in the LBL "++" ++...+ XROM LB sequence as described in the instructions for LB . Then XEQ "XLB" and supply two XROM numbers in response to the prompt. For instance for XROM 10, 00 key in 10 ENTER↑ 0. Press R/S to calculate and load two bytes. When the next prompt for XROM numbers appears you can either enter another pair of numbers or press R/S without an input to terminate the byte-loading process. The usual prompt "SST, DEL 00p" will be given.

Example 2: If you change line 23 of "XLB" (Example 1) from CF 09 to CF 08, pressing R/S without an input will not terminate the byte loading. Instead, the CF 08 instruction switches to the manual LB operation, allowing additional bytes to be loaded from the keyboard.

COMPLETE INSTRUCTIONS FOR -B

These routines allow bytes to be loaded under the control of your own program . The general rules for their use are as follows:

1. In the program that you are writing which controls the loading of bytes, put the instruction XROM L- . This initializes the byte-loading process and then (instead of prompting for Byte #1) returns to your control program.

2. Have your control program calculate or otherwise place each byte (only decimal allowed in this version) in the X-register, and put an XROM -B in your program to load that byte. Flags 22 and 23 are ignored, as well as whether the calculator is in ALPHA or non-ALPHA mode. The call to routine -B causes one byte to be loaded and then returns to your control program.

3. To terminate the byte-loading process, put the instructions CF 09, XROM -B in your control program. Executing the routine -B with Flag 09 cleared will cause no additional byte to be loaded, but rather a termination of the byte loading, in this case *not* returning to your control program, but ending with a "SST, DEL 00*p*" prompt. Executing -B with Flag 08 cleared will switch from automatic to manual byte loading, allowing more bytes to be loaded directly from the keyboard.

4. Before running your control program, check for size 12, and make room in program memory where you want the bytes to be loaded in exactly the same way as when using the prompting version LB . That is, key in (in PRGM mode) LBL"++", a string of +'s, and XROM LB .

5. Switch out of PRGM mode and instead of pushing R/S to start the byte loader, execute your own control program. Then sit back while your program (if correctly written) calculates, prompts for, or otherwise creates and loads each byte.

6. Execution will terminate with the "SST, DEL 00*p*" prompt, whereupon you can perform the "cleanup" operations just as with the ordinary LB program.

7. If you want your control program to correct a byte that it previously loaded, have it enter a negative number in X and execute -B to get rid of the last-entered byte.

8. Your control program is welcome to make use of any of the contents of registers 06-11 (see above), as long as it doesn't change any of these registers.

WARNING: Don't execute -B (or let your program do it) without having first initialized the process by executing L- ! A few flag and other safeguards have been incorporated, but executing -B by itself *could* cause MEMORY LOST or destruction of existing programs.

When used properly, L- and -B can be very powerful, ultimately allowing one to write a program which writes programs! A somewhat less exotic application is a byte loading program which allows bytes to be scanned in by a wand instead of keyed in.

APPLICATION PROGRAM 1 FOR -B

The following program "LBW" (Load Bytes With Wand) allows bytes to be loaded by scanning 2-byte paper keyboard (type 5) barcodes. Only the second byte of the barcode is loaded into program memory, but in order to avoid scanning errors the entire barcode is checked for checksum consistency. Using this program along with a barcode hex table (such as in PPC CJ, V7N6P25-26) and HP's Wand Paper Keyboard, one can rapidly scan in the bytes to be loaded in a manner which for many functions is similar to the normal use of the paper keyboard.

For example, the synthetic instruction X<> can be obtained by scanning X<> in the Paper Keyboard (which will supply the correct prefix) and then byte 78 (hex) in the hex table for the postfix, and TONE 26 can be obtained by scanning TONE from the paper keyboard and byte 26 (decimal) from the hex table. For alpha characters, the barcode hex table can be used, but not the alpha character codes in the Paper Keyboard (or in the character table of PPC CJ, V7N6P23) which use a different format for encoding the character.

Instructions for using "LBW" are as follows:

1. Insert LBL ++, a string of +'s, and XROM LB in the desired part of program memory, just as when using LB .

2. Switch out of program mode and XEQ "LBW". (Note: SIZE 012 or greater is required. If you get the insufficient SIZE message, re-size the calculator

APPLICATION PROGRAM FOR: **-B**

```
01♦LBL "LBW"          36 GTO 01
02 XROM "L-"          37♦LBL 14
03♦LBL 01             38 SIGN
04 FIX 0              39 X≠Y?
05 CF 29              40 GTO 14
06 "W: "              41 90
07 ARCL 06            42 RCL 01
08 "⊢ OF "            43 X=Y?
09 ARCL 07            44 GTO 10
10 XROM "VA"          45 189
11 TONE 7             46 X≠Y?
12 .                  47 GTO 12
13 CF 22              48♦LBL 03
14 XROM 27,05         49 -1
15 FS? 22             50 GTO 11
16 GTO 11             51♦LBL 14
17 2                  52 X<>Y
18 X≠Y?               53 X=0?
19 GTO 14             54 GTO 10
20 RCL 01             55 5
21 16                 56 X<Y?
22 XROM "QR"          57 GTO 13
23 RCL 02             58♦LBL 12
24 +                  59 "BAR/CKSM ERR"
25 X≠0?               60 XROM "VA"
26 15                 61 TONE 1
27 X≠0?               62 GTO 01
28 MOD                63♦LBL 10
29 X=0?               64 CF 09
30 X<> L              65 GTO 11
31 X≠Y?               66♦LBL 13
32 GTO 12             67 "LOAD ABORT"
33 RCL 02             68 TONE T
34♦LBL 11             69 PROMPT
35 XROM "-B"          70 GTO 13
                      71 .END.
```

NOTE: This program also appears under L- .

and then key in XEQ "LBW" again to restart the process. Just pushing R/S after re-sizing will cause the ordinary **LB** byte loading version to be initiated instead of the wand version.

3. At each prompt "W: N OF M", scan in the appropriate 2-byte barcode (the WNDSCN command is in effect here). After verifying the checksum, the second byte of the barcode will be loaded.

4. A decimal entry can be made directly from the keyboard by clearing the "W: N OF M" prompt (using ←), making the entry, and pushing R/S. Flag 22 is used to detect such an entry. Afer loading the byte, the program will resume with the "W: N+1 OF M" prompt. Hexadecimal entries are not provided for in this program however.

5. To correct an entry, either (a) scan the 1-byte ←barcode, or (b) clear the prompt and XEQ 03, or (c) clear the prompt, enter a negative number, and push R/S. Method (a) can be used to conveniently clear up to 3 bytes by making up to 3 scans at once and waiting while they are processed one by one.

6. During the prompt for a new byte, X=0 while Y= decimal value of previous byte. If you wish to clear the prompt to check the previous byte value, make elementary calculations, etc., push XEQ 01 afterward to get a re-prompt before continuing with the loading.

NOTES

7. To terminate the byte-loading process, either (a) scan the one-byte . (decimal point) barcode, or (b) push R/S twice. Then follow the usual "clean-up" procedures as with **LB** . The loading process will also terminate itself automatically after the maximum number of bytes is reached.

8. If you have accidentally terminated and wish to add more bytes or make corrections, push GTO 03 R/S or GTO 01 R/S (rather than XEQ 03 or XEQ 01, which would disable the return to the "LBW" program).

9. Scanning any 1-byte barcode other than ← or . or any barcode of 3 to 5 bytes will cause the message "BAR/CHKSM ERR" and a re-prompt. The same applies to a 2-byte barcode whose checksum does not check. However, scanning a 6-byte or longer barcode will cause vital information in R06-R11 to be wiped out, so in such a case the whole process is terminated with a "LOAD ABORT" message.

To give a brief analysis of the program:

Lines 01-23 initialize the loading process, and lines 03-14 set up the prompt and execute the WNDSCN command. Lines 15-16 detect an entry from the keyboard and branch to lines 34-35 to load the byte (or backup, if the entry is negative). Otherwise a scan with the wand is assumed to have occured, in which case WNDSCN causes the number of bytes to be in X and the decimal byte values in R01-R0k. If k≠2, a branch is made (lines 17-18) to line 37; otherwise the 4-bit wraparound checksum of the last 3 nybbles is calculated and compared with the first nybble (lines 20-32). A mismatch causes a branch at line 32 to LBL 12 (line 58) where the error message is given; otherwise the second byte of the barcode is recalled and loaded (lines 33-35) and we start over (line 36).

If k≠2 then we had branched to line 37, after which we check for k=1, and if true we check whether the one byte was 90 decimal (5AH, the decimal print code) or 189 (BDH, the back-arrow code) and branch accordingly, otherwise branching to the error message. Lines 51-57 deal with the case where k is neither 1 nor 2; if k=0 then no scan has taken place and it is assumed that R/S R/S was pushed, so we branch to line 63 and initiate the termination procedure by clearing flag 09. If k>5 we branch to line 66 to produce the "LOAD ABORT" message. For other values of k the "BAR/CHKSM ERR" message is produced in lines 58-61, and line 62 branches to a re-prompt.

*The checksum can be calculated by adding up the decimal values of the nybbles; if the result is zero proceed no further. Otherwise take the result mod 15 and if the result of that is zero, change it to 15. In the present case, we are concerned with the last nybble (call it n2) of R01 and both nybbles (call them n3 and n4) of R02, and since n2 + n3 + n4 is equivalent to n2 + 16*n3 + n4 when taken mod 15, it is necessary to decompose the byte in R02 (=16*n3 + n4) into its separate nybbles before adding. Routine **QR** is used, however, to decompose the byte in R01 into its separate nybbles n1 and n2; n1 is the number to be compared with the calculated checksum.

APPLICATION PROGRAM 2 FOR **-B**

As an example of a program which writes programs, the following program, "COMP", composes random music by generating a program consisting of tone instructions selected at random from tones 0 through 127 using routine **RN** to generate the random numbers. To use it, initialize the desired section of program memory with the usual LBL ++, string of +'s, and XROM **LB** , and then go into non-PRGM mode, make sure the SIZE is at least 012, and execute "COMP". The program will prompt for a seed; enter any number and push R/S. The tone instructions will be loaded into program memory until there is no room left, whereupon the usual "SST, DEL OOP" termination will occur. After performing the usual cleanup operations you can execute your newly composed program and hear the music.

This program can be directly compared with Application Program 2 for **TN** , "MUS", which generates and plays the tones in "real time". The generation of the random numbers is exactly the same for the two programs (see the description of "MUS" under **TN** for an explanation), and the tones produced by "MUS" and "COMP" for a given initial seed, will be the same up to the point where the latter runs out of memory space. "MUS" has the advantage of producing tones indefinitely with no initial compilation time, but the listener must put up with the approximately 2-second delay between tones, making the "music" rather tedious. "COMP" requires an initial compilation time (3-4 minutes to generate a 49 tone sequence) and the length of the piece is limited by the number of +'s initially put into program memory, but once the compilation is done the music can be played with no intertone delays. Thus, the results of "COMP" (though they may not become instant hits) are likely to be much more satisfying to the listener.

Lines 02-13 of "COMP" initialize the random numbers (see "MUS" under **TN**), store frequently used constants, and initialize the byte-loading procedure. Lines 14-21 take the integer part of R07, which is the maximum number of bytes that can be loaded, determine whether it is even or odd, and load a null byte (line 21) if the number is odd. This ensures that there is an even number of bytes left over that can be loaded so we can simply load tone instructions repeatedly (at 2 bytes per instruction) until we run out of bytes, at which time **-B** will terminate the loading automatically, and the termination will not be in the middle of an instruction. Lines 22-30 form the tone-loading loop in which we first (lines 23-24) load byte 159 (decimal) corresponding to the TONE prefix, then obtain a random number whose integer part is uniformly distributed from 0 to 127 (lines 25-28) and use it for the postfix byte (line 29).

As an aid to the mass production of music (or other byte loading operations) one can record on a single track of a card the following 112-byte program: LBL ++, a string of 104 +'s, XROM **LB** . (When recording and reading this card there will be a prompt for side 2 which you can ignore and clear). Reading this card and using any of the versions of the byte loader will always allow exactly 98 bytes to be loaded, on our present case allowing 49 tones. The final 49-note piece will then fit onto one track of a card with a few bytes to spare for labels, etc.

As an example of HP-41 generated music, the author found particularly nice the 49-note piece (which coincidentally, takes just 49 seconds to play) obtained by using the card described in the last paragraph and inputting a seed of 4; the initial compilation took 3.25 minutes. If however, the user is not so enthralled by this particular composition, he has plenty of others to choose from. And whether or not he would agree that such music is a manifestation of the true soul of the HP-41, it is undeniable that all of this is an interesting example of calculator composed music, programs that generated programs, and the art of synthetic programming in general. Further refinements could include, for example, weighting factors to favor (say) the short duration tones, and even some "rules of composition" to produce particular musical effects.

BAR CODE ON PAGE 483

APPLICATION PROGRAM FOR: **-B**

```
01♦LBL "COMP"
02 "SEED?"
03 PROMPT
04 ABS
05 LN
06 ABS
07 FRC
08 STO 00
09 159
10 STO 04
11 128
12 STO 05
13 XROM "L-"
14 RCL 07
15 INT
16 2
17 MOD
18 1
19 -
20 X=0?
21 XROM "-B"
22♦LBL 01
23 RCL 04
24 XROM "-B"
25 CLX
26 XROM "RN"
27 RCL 05
28 *
29 XROM "-B"
30 GTO 01
31 .END.
```

LINE BY LINE ANALYSIS OF **-B**

See **LB** .

CONTRIBUTORS HISTORY FOR **-B**

L- and **-B** were conceived and written by Roger Hill (4940) as an integral part of the ROM version of **LB** .

Routine Listing For: —B

```
01♦LBL 00
02 STOP
03 GTO "++"

78♦LBL 06
79 STO 11
80 CLA

81♦LBL 07
82 ASTO 08
83 X<>Y
84 ISG 06

85♦LBL 15
86 SF 09
87 FS? 08
88 RTN
89 CF 22
90 CF 23
91 FIX 0
92 CF 29
93 "#"
94 ARCL 06
95 "├ OF "
96 ARCL 07
97 "├?"
98 XROM "VA"
99 TONE 7
100 STOP
101 FS? 48
102 GTO 14
103 FC? 22
104 GTO 19
105 GTO 08

106♦LBL 14
107 FC? 23
108 GTO 19
109 XROM "XD"

110♦LBL 08
111 X<0?
112 GTO 03
113 ENTER↑
114 CLA
115 ARCL 08
116 XROM "DC"
117 RCL 06
118 X<=0?
119 GTO 10
120 7
121 MOD
122 X≠0?
123 GTO 07
124 X<>Y
125 RCL 09
126 RCL 10
127 X<> c
128 RCL [
129 STO IND Z
130 X<>Y
131 X<> c
132 R↑
133 RCL 08
134 DSE 09
135 GTO 06
136 ISG 09
137♦LBL 20
138 CF 09
139 CLX
140 X<>Y
141 RCL 07
142 FRC
143  E3
144 *
145 AOFF
146 FIX 0
147 "SST, DEL 00"
148 ARCL X
149 FIX 3
150 XROM "VA"
151 BEEP
152 GTO 00

203♦LBL "-B"
204 FC? 08
205 GTO 15
206 FS? 09
207 GTO 08

208♦LBL 19
209 RCL 06
210 X<=0?
211 GTO 10
212 CHS
213 ISG X
214 7
215 MOD
216 X=0?
217 GTO 14
218 CLA
219 ARCL 08

220♦LBL 11
221 "├♦"
222 DSE X
223 GTO 11
224 X<> [

225♦LBL 14
226 RCL 09
227 X<>Y
228 RCL 10
229 X<> c
230 X<>Y

231♦LBL 12
232 STO IND Z
233 CLX
234 DSE Z
235 GTO 12
236 RDN
237 X<> c
238 RDN
239 GTO 20
```

FURTHER ASSISTANCE ON —B

Call William Cheeseman (4381) at (617) 235-8863.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,24 —B SIZE: 012

Stack Usage:
- 0 T: PREV. Y IF FL 09 SET
- 1 Z: PREV. Y IF FL 09 SET
- 2 Y: ALTERED
- 3 X: SEE * IN COMMENTS
- 4 L: USED

Flag Usage:
- 04: NOT USED
- 05: NOT USED
- 06: NOT USED
- 07: NOT USED
- 08: USED
- 09: CLEAR TO TERMINATE BYTE LOADING
- 10: NOT USED
- 29: CLEARED UPON TERMINATION OF BYTE LOADING

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

ALL USED

Other Status Registers:
- 9 Q: NOT USED
- 10 ├: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED BUT RESTORED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED IF FLAG 9 SET: FIX 3 IF FLAG 9 CLEAR

Angular Mode: UNCHANGED

Unused Subroutine Levels: 4

ΣREG: UNCHANGED

Data Registers:

R00: ONLY REGISTERS 6-11 ARE USED

R06: BYTE NUMBER

R07: m.00p00q

R08: PARTIAL REGISTER OF BYTES

R09: INDEX FOR BYTES STORAGE

R10: Rc FOR LOWERED CURTAIN

R11: PREVIOUSLY-STORED 7 BYTES

Global Labels Called:

Direct	Secondary
PART OF LB	DC
	OM
	QR
	VA
	PART OF GE

Local Labels In This Routine:

11
12
14
19

Execution Time: 1.6 seconds for FLAG 9 set.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	LB
Bytes In RAM:	66
Registers To Copy:	71

Other Comments:

*Unchanged if flag 9 set; p.00q if flag 9 clear.

1K - FIRST KEY ASSIGNMENT

1K is a subroutine version of **MK**. It allows your programs to set up their own key assignments without any user intervention required. **1K** performs the same tasks that **MK** does on the first key assignment, except that prompting is suppressed. The three inputs--prefix, postfix, and user key code--are presumed to be in Z, Y, and X when **1K** is called. Error messages will be generated if the key is nonexistent or occupied, unless flag 25 was set, in which case the error message is left in the alpha register upon RTN to the calling program. Like **MK**, **1K** does not pack the key assignment registers in the interest of saving execution time.

Example 1: Suppose your program requires the assignment of RCL b to the 1/x key in order to allow the user to enter program pointer data. You would simply insert the steps

```
144
124
 12
XROM 1K
```

at the appropriate point in your program. The SIZE must be at least 012 and registers 06-11 will be used as for **MK**.

BACKGROUND FOR 1K

See **MK**.

COMPLETE INSTRUCTIONS FOR 1K

These allow you to make key assignments under program control.

1. Make sure (or have your program make sure) that the size is at least 12.

2. If you want your control program to automatically make one key assignment, have it place the prefix, postfix, and keycode in Z, Y, and X respectively, and then have it call **1K** as a subroutine (XEQ **1K**). The key assignment will be made and control returned to your program. (The display of the number of free registers is bypassed.) If an error occurs, then the program will stop with the appropriate error message *unless* you have set Flag 25 before calling **1K**, in which case an error will cause Flag 25 to be cleared and control returned to your program without the assignment being made (similarly to the machine use of Flag 25), but with error message in the alpha register in case you want to know what kind of error it was.

3. To make several key assignments under program control, the first assignment can be made by XEQ **1K** and each additional assignment made by XEQ **+K**). (Each routine is called, of course, after placing the appropriate data in Z, Y, and X.) Routine **+K** does the same as **1K** except that it bypasses the register counting (if Flag 20 is set), enabling additional key assignments to be made much faster. However, if Flag 20 is clear, then **+K** does include the register count and is then identical to **1K**. (Flag 20 is set, by the way, after a register count.) If your program is such that it is not convenient to call **1K** for the first assignment and **+K** for the rest (e.g. involving a loop where the subroutine call appears only once in the program), the **+K** may be used for *all* of the key assignments as long as you include the two instructions SF 07, CF 20 before calling **+K** the first time. (SF 07 selects the non-prompting mode.)

4. Routines **1K** and **+K** can also be executed from the keyboard, after placing the required inputs in the stack. When routine **+K** is executed, it will continue key assignments using the prompting version if **MK** was previously executed (Flag 07 clear), or the non-prompting version if **1K** was previously executed (Flag 07 set).

WARNING: Don't use **+K** by itself (without either **MK** or **1K** having been executed) unless you are sure that Flag 20 is clear, and you have checked Flag 07 to make sure you are getting the version you want. The same warning (to make sure Flag 20 is cleared) applies if the size has been changed, key assignments have been manually added or deleted, key assignments have been packed, or R_{09}-R_{11} have been altered.

MORE EXAMPLES OF 1K

Example 2: **1K** can be executed from the keyboard to save a second or so over **MK**. You must remember to supply input before executing **1K** and to set SIZE $\leq$ 012. If you get NONEXISTENT (due to insufficient SIZE), resize and start over.

APPLICATION PROGRAM 1 FOR 1K

The following program, "RSB", takes the number in X as a user keycode and assigns RCL b to the unshifted key and STO b to the shifted key. If either key is already occupied the program will stop for you to either clear the pre-existing assignment or enter an alternate key code (in the later case, altering the code for the unshifted key will also alter the shifted key.) It is assumed that the size is 12 or greater.

```
LBL "RSB"     (Recall and Store b)
CF 25
144
X<>Y
124
X<>Y
XROM 1K
145
RCL 07
RCL 08
CHS
XROM +K
END
```

(30 bytes)

APPLICATION PROGRAM 2 FOR 1K

The synthetic instruction RCL b is often useful to have as a key assignment, as it allows recalling the program pointer without having altered it, and this can be used in byte-counting and other applications. The following program, ARB, assigns RCLb to the first unused, unshifted key in the top row, working from left to right. If all of these keys already have assignments, "ROW FULL" is displayed with a warning tone. To use, just XEQ "ARB", no input being necessary. If an unoccupied key is found, the assignment will be made and the message "RCL b KEY..." will appear to show which key was used.

APPLICATION PROGRAM FOR: 1K

```
01♦LBL "ARB"          20 STOP
02♦LBL 00             21 GTO 00
03 CF 25              22♦LBL 02
04 35.00008           23 X<> d
05 STO 11             24 144
06 11                 25 FIX 0
07 RCL ⊢              26 CF 29
08 X<> d              27 124
09♦LBL 01             28 R↑
10 FS? IND Z          29 XROM "1K"
11 ISG Y              30 "RCL b KEY"
12 GTO 02             31 ARCL 08
13 DSE Z              32 XROM "VA"
14 GTO 01             33 SF 27
15 X<> d              34 BEEP
16 CLST               35 END
17 "ROW FULL"
18 XROM "VA"
19 TONE 3
   < 83 >
```

Line 05 is to check for size at least 12. The ⊢-register (shown as ᵀ on the printer) is recalled and put into the flag register, and in the loop consisting of lines 09-14 flags 35, 27, 19, 11, and 03 are checked, corresponding to keys 11, 12, 13, 14, and 15. The keycode in Y is incremented (line 11) each time a set flag is encountered, and line 12 is skipped; if a clear flag is found line 11 is skipped and line 12 takes us out if the loop to LBL 02 (line 22), where the assignment is made to the key whose code was in Y at the time of leaving the loop, the message is displayed, and the calculator is set to USER mode by setting flag 27. If the loop is terminated without having found a clear flag, the "ROW FULL" message is displayed. (Line 19 is TONE 83, for which any other tone or sequence of tones can be substituted.) The user may then clear one or more keys and push R/S to start the program again.

A slightly shorter version can be written which tries to make an assignment to each key (with flag 25 set) until it succeeds, but it takes considerably more time to execute if several tries need to be made, even if the register check is bypassed after the first try.

Routine Listing For: 1K

```
08♦LBL 01            76 MOD              146 STO IND Z
09 XROM "LF"         77 8                147 X<>Y
10 STO 09            78 *                148 X<> c
11 E                 79 ENTER↑           149 CLST
12 +                 80 CF 08            150 CLA
13 X<>Y              81 LASTX            151 FC? 10
14 STO 10            82 FS? 09           152 ISG 09
15 ASTO 11           83 ST+ Y            153 SF 20
16 DSE Y             84 R↑               154 FS? 07
17 GTO 07            85 INT              155 RTN
18 SF 20             86 X≠0?             156 FS? 20
19 FC?C 09           87 X>Y?             157 GTO 03
20 GTO 13            88 GTO 08           158 "DONE, NO MORE"
                     89 R↑               159 SF 09
21♦LBL 02            90 +                160 GTO 14
22 RCL 09            91 ST+ Z
23 XEQ 11            92 X<>Y             161♦LBL 07
24 "REG FREE: "      93 X<=Y?            162 "NO ROOM"
25 RCL d             94 CLX              163 CF 20
26 FIX 1             95 X≠0?             164 CLST
27 ARCL Y            96 SF 08
28 STO d             97 +                165♦LBL 14
29 XROM "VA"         98 36               166 FS?C 25
30 TONE 6            99 -                167 RTN
31 PSE               100 X>0?            168 XROM "VA"
                     101 GTO 08          169 TONE 7
32♦LBL 03            102 FC? 09          170 TONE 3
33 "PRE↑POST↑KEY"    103 RCL ⊢           171 STOP
34 CLST              104 FS? 09          172 GTO 01
35 XROM "VA"         105 RCL e
36 TONE 7            106 FC? 08          173♦LBL 08
37 STOP              107 GTO 14          174 "NO SUCH KEY"
38 GTO 14            108 STO [           175 GTO 14
                     109 "⊢*"
39♦LBL "1K"          110 X<> [           176♦LBL 09
40 CF 20                                 177 X<> d
41 SF 07             111♦LBL 14          178 "KEY TAKEN"
                     112 X<> d
42♦LBL "+K"          113 FS? IND Y       179♦LBL 14
43♦LBL 14            114 GTO 09          180 CF 09
44 STO 08            115 SF IND Y        181 CF 20
45 RDN               116 X<> d           182 FS?C 25
46 STO 07            117 FC? 08          183 RTN
47 RDN               118 GTO 14          184 XROM "VA"
48 STO 06            119 STO [           185 TONE 3
49 CF 09             120 ARCL 10         186 PSE
50 RCL 10            121 X<> \           187 "KEYCODE?"
51 SIGN                                  188 CLST
52 FS? 20            122♦LBL 14          189 RCL 08
53 X≠0?              123 FC? 09          190 XROM "VA"
54 GTO 01            124 STO ⊢           191 TONE 7
                     125 FS?C 09         192 STOP
55♦LBL 13            126 STO e           193 STO 08
56 RCL 08            127 "**"            194 GTO 01
57 INT               128 FS? 10
58 X=0?              129 ARCL 11         197♦LBL 11
59 FS? 07            130 X<> Z           198 INT
60 FC?C 20           131 RCL 07          199 LASTX
61 GTO 02            132 RCL 06          200 FRC
62 X<0?              133 XROM "DC"       201 E3
63 SF 09             134 XROM "DC"       202 *
64 ABS               135 XROM "DC"       203 X<>Y
65 STO Z             136 FS?C 10         204 .5
66 44                137 GTO 14          205 FC? 10
67 -                 138 "⊢♦♦♦"          206 SIGN
68 ABS               139 ASTO 11         207 -
69 2                 140 SF 10           208 -
70 X<Y?                                  209 END
71 DSE T             141♦LBL 14
72 R↑                142 RCL 09
73 STO Y             143 RCL 10
74 E1                144 X<> c
75 ST/ Z             145 RCL [
```

TECHNICAL DETAILS

XROM: 10,02	1K	SIZE: 012

Stack Usage: ALL CLEARED 0 T: USED 1 Z: PREFIX INPUT 2 Y: POSTFIX INPUT 3 X: KEYCODE INPUT 4 L: USED	**Flag Usage:** 04: NOT USED 05: NOT USED 06: NOT USED 07: SET 08: USED 09: USED 10: USED 20: REGISTER CHECK DONE 25: CLEARED
Alpha Register Usage: 5 M: 6 N: ALL CLEARED 7 O: 8 P:	
Other Status Registers: 9 Q: NOT USED 10 ⊢: ALTERED 11 a: NOT USED 12 b: NOT USED 13 c: USED BUT RESTORED 14 d: USED BUT RESTORED 15 e: ALTERED	**Display Mode:** UNCHANGED **Angular Mode:** UNCHANGED **Unused Subroutine Levels:** 3
ΣREG: UNCHANGED **Data Registers:** R00: ONLY REGISTERS 6-11 ARE USED R06: PREFIX R07: POSTFIX R08: KEYCODE R09: INDEX FOR STORAGE R10: Rc FOR LOWERED CURTAIN R11: FIRST ASSIGNMENT OF PAIR	**Global Labels Called:** Direct: LF, DC, PART OF MK Secondary: E?, OM, 2D, PART OF GE **Local Labels In This Routine:** 07 08 09 11 13 14
Execution Time: 8.6 - 13.5 seconds. (0-30 Existing Assignments)	
Peripherals Required: NONE	
Interruptible? NO Execute Anytime? NO Program File: MK Bytes In RAM: 312 Registers To Copy: 61	**Other Comments:**

LINE BY LINE ANALYSIS OF 1K

See MK.

CONTRIBUTORS HISTORY FOR 1K

The programmable key assignment capabilities of the PPC ROM, consisting of the 1K and +K programs, were conceived and implemented by Roger Hill (4940).

FURTHER ASSISTANCE ON 1K

Call Roger Hill (4940) at (618) 656-8825.

NOTES

APPENDIX A –
ADVANCED APPLICATIONS OF LR/SR & HD/UD

HANOI TOWER PUZZLE GENERALIZED

Given: $\underline{m}$ pegs, $\underline{n}$ discs of varying size stacked in order of size (large on the bottom, small on the top) on peg 1.

Problem: In the smallest number of moves, one disc at a time, in such a way that a disc is never placed on top of a smaller one, move the $\underline{n}$ discs (similarly stacked) from peg 1 to peg m.

A brief glance at the original 3-peg problem (Tower of Hanoi) will prove helpful. The crux of the solution to this simplest version is the need to uncover the bottom disc, which in turn leads to the need to transfer $\underline{n-1}$ discs to peg 2. This perspective leads to a repeated reduction by one of the number of discs to be moved, until we are led to the need to move only one disc. The immediately preceding problem was the need to move 2 discs; and the solution has become: first move the top disc to the other peg, then move the bottom disc to the target peg, and finally move the top disc again. The top disc was moved twice. If there were 3 discs to move, the top disc would be moved twice in loading the alternate peg, and then twice more in unloading to the target peg. The inductive argument shows that if there are $\underline{n}$ discs to move, the top disc undergoes 2^{n-1} moves, the disc below undergoes 2^{n-2} moves, etc., while the bottom disc requires only $2^{n-n} = 1$ move. There are easier ways of establishing the total number of moves ($1+2+2^2+\ldots+2^{n-1}=2^n-1$), but this way of looking at it has value for resolving the problem with more than 3 pegs.

We need to explicitly note some parallel features of the m-peg version of this puzzle.

(A) If using $\underline{m}$ pegs, we have $\underline{m-2}$ pegs (pegs 2,3,..., m-1) to temporarily hold the top $\underline{n-1}$ discs while we move the bottom disc to peg m.

(B) Unpacking the top $\underline{n-1}$ discs can be viewed as $\underline{m-2}$ subtasks to be performed sequentially:

(1) Using all $\underline{m}$ pegs, first load n_m discs onto peg 2.

(2) Using $\underline{m-1}$ pegs (by the rules peg 2 can't be used), load n_{m-1} discs onto peg 3.

(3) Using $\underline{m-2}$ pegs (pegs 2 and 3 can't be used), load n_{m-2} discs onto peg 4.

⋮

(m-2) Using 3 pegs (pegs m, 1, and m-1), load n_3 discs onto peg m-1.

(C) After moving the bottom disc to peg m, unload the substacks in a sequence opposite to the loading:

(1) Using 3 pegs (m-1, 1, and m) transfer the n_3 discs on peg m-1 to peg m.

(2) Using 4 pegs (m-2, 1, m-1, and m) transfer the n_4 discs on peg m-2 to peg m.

⋮

(m-2) Using all $\underline{m}$ pegs transfer the n_m discs on peg 2 to peg m.

(D) Note that unloading a peg to the target peg entails the same number of moves as loading the peg from the original stack.

Again we can show that the number of moves any disc undergoes in arriving at its final location is a power of 2, but the reasoning is more complicated than when we assume that m=3.

Suppose we want to know the number of moves required to place the top disc of a stack of $\underline{n}^{(0)}$ discs when we're using $\underline{m}$ pegs. By observation B, we know that if $\underline{n}^{(0)} > 1$, our first subtask is to move $\underline{n}^{(1)}$ discs to peg 2, where $\underline{n}^{(1)} < \underline{n}^{(0)}$. If $\underline{n}^{(1)} > 1$, we can proceed to the first subtask of the first subtask, and that would entail the movement of $\underline{n}^{(2)}$ discs to some peg, where $\underline{n}^{(2)} < \underline{n}^{(1)}$. By this recursion, we arrive at the transfer of the top disc to some peg. By observation D, unraveling this recursion leads to repeated doubling of the number of moves undergone by the top disc. Of course, when $\underline{m} > 3$, the total number of moves of the top disc of a stack containing $\underline{n}$ discs is less than 2^{n-1}. The disc below will require either as many or half as many moves. Let's proceed to make this more definite.

Let $Z_n(m)$ = number of moves required to transfer n discs using m pegs. Clearly $Z_1(m) = 1$. How does $Z_{n+1}(m) - Z_n(m)$ behave?? Certainly $Z_2(m) - Z_1(m) = 2$, since the top disc has to be placed on and removed from an intermediate peg. Obviously, in fact, $Z_{n+1}(m) - Z_n(m)$ remains 2 up to $n = m-2$, there being $\underline{m-2}$ intermediate pegs available. At this point we're out of intermediate pegs, so at least one disc will require more than one intermediate resting place. By observation D and the preceding discussion, that disc will require 4 moves. As $\underline{n}$ increases, $Z_{n+1}(m) - Z_n(m)$ eventually becomes 8, still later 16, etc. To be more specific, we need an appropriate recursive relationships.

Let $Q(m,e)$ = the maximum $\underline{n}$ such that $Z_n(m) - Z_{n-1}(m) = 2^e$. In order to extend this definition to $Q(m,0)$, let $Z_0(m) = 0$. Suppose we know $Q(\mu,e)$ for $\mu = 3$ to $\underline{m}$, and let $\underline{N} = 1 + \Sigma\, Q(\mu,e)$, (where the summation is from $\mu=3$ to $\mu=\underline{m}$), be the number of discs we will transfer using $\underline{m}$ pegs. Using the strategy noted in observations $\underline{B}$ and C:

(1) Transfer Q(m,e) discs to peg 2 from peg 1.

(2) Transfer Q(m-1,e) discs to peg 3 from peg 1.

APPENDIX A CONTINUED ON PAGE 37

2D - DECODE 2 BYTES TO DECIMAL

2D decodes each of the "last" (rightmost in the display) two bytes contained in the X-register into their decimal equivalents. The decimal equivalent of the penultimate byte is left in the X-register; that of the last byte is left in register M. **2D** is especially handy for decoding program pointers, which consist of two right-justified bytes.

Example 1: Use **2D** to decode the last two bytes of a number.

DO:	SEE:	RESULT:
1.123456789E-10	153	$153_{10}=99_{16}$ was the next to last byte in the X-register, consisting of the last mantissa digit "9" and the digit "9" used to designate the negative exponent sign.
RCL M	144	$144_{10}=90_{16}$ was the 1st byte, since negative exponednt digits mn are coded as (100-mn).

COMPLETE INSTRUCTIONS FOR 2D

The input for **2D** consists of the last two bytes, b_1 and b_0, of the hexadecimal code contained in the X-register. Therefore, prior to execution of **2D**, the user must ensure that the two bytes to be decoded are in those locations. Upon execution of the routine, the decimal equivalent (0 - 255, as listed in the "combined hex table") of b_1 will be left in the X-register. That of b_0 is left in the alpha register M, from which it may be retrieved with the synthetic operation RCL M.

2D uses the entire alpha register. The original contents of stack registers Y and Z are preserved in Y and Z, but L, X and T are lost. Therefore, in order to preserve the code in the X-register that is partially decoded by **2D**, execution of the routine should be preceded by an ENTER↑.

MORE EXAMPLES OF 2D

Example 2: Use **2D** to locate the HP-41C/V internal ROM address associated with the SIZE function. That address is left in the last two bytes of register Q when SIZE is assigned to a user key (see PPCCJ, V8N2P47). First, have the synthetic functions RCL Q and RCL M assigned to keys, using **MK** if necessary.

	DO:	SEE:	RESULT:
1.	FIX 9; ASN SIZE to any key; RCL Q	0.0000040-08	Contents of register Q (hex.)
2.	FIX 0; XEQ 2D	18.	penultimate byte of Q (dec.)
3.	RCL M	146.	last byte of Q (dec.)

Since $18_{10} = 12_{16}$ and $146_{10} = 92_{16}$, the desired ROM address (in hexadecimal digits) is 1292.

APPLICATION PROGRAM 1 FOR 2D

The routine "2BD" given here offers a faster alternative to the PPC ROM routine **BD** for decoding 2-digit numbers in base B, 2≤B≤36, to decimal. The heart of the routine is in the use of **2D**, at line 07, to convert an alphanumeric character pair keyed in by the user into their respective decimal values as given in the "combined hex table." Lines 08-23 of the routine transform those values so that the alpha characters "A" through "Z" are represented by 10 through 35. The base B (in decimal) is keyed in by the user prior to execution of the routine. Multiplication of the penultimate digit by B, and adding the product to the last digit, complete the routine.

INSTRUCTIONS FOR 2BD:

Key in the base B, 2≤B≤36, and XEQ "2BD". Execution will halt in ALPHA mode with the prompt "CODE?" Key in the 2-digit number to be decoded as an alphanumeric pair, with the digits 10, 11, ..., 35 represented by A, B, ..., Z. Press R/S to see the decimal equivalent. To repeat with the same B, press R/S; to repeat with a new B, key in B and press R/S.

Example: Find the decimal equivalents of FF_{16}, ZZ_{36}, and 99_{36}:

	DO:	SEE:	RESULT:
1.	16 XEQ "2BD"	"CODE?"	Prompt for alphanumeric entry, B=16
2.	FF R/S	255	$FF_{16} = 255_{10}$
3.	36 R/S	"CODE?"	Prompt, B=36
4.	ZZ R/S	1295	$ZZ_{36} = 1295_{10}$
5.	R/S	"CODE?"	Prompt, B=36 again
6.	6. 99 R/S	333	$99_{36} = 333_{10}$

APPLICATION PROGRAM 2 FOR 2D

```
01 LBL "2BD"
02 "CODE?"
03 AON
04 STOP
05 AOFF
06 RCL M
07 XROM 2D
08 51
09 ST-M
10 -
11 7
12 X<Y?
13 ST-Y
14 RCL M
15 X<>Y
16 X<Y?
17 ST - Y
18 CLX
19 3
20 ST+Z
21 +
22 X<>Y
23 R↑
24 *
25 +
26 R↑
27 VIEW Y
28 END
```

Further examples of the use of **2D** may be found in the PPC ROM routines **RT** (Return Address to Decimal), **PD** (Program Pointer to Decimal), and **E?** (End Finder), all of which use **2D** as a subroutine.

Routine Listing For:	2D
94♦LBL "2D"	108 *
95 "*"	109 ST+ [
96 X<> [	110 X<> \
97 X<> \	111 RCL]
98 ASHF	112 INT
99 "├♦↓♦'♦♦"	113 HMS
100 X<> [	114 *
101 X<> \	115 RCL]
102 X<> [	116 +
103 "├♦♦B"	117 E1
104 RCL [	118 ST* [
105 INT	119 *
106 +	120 X<> [
107 RCL \	121 RTN

LINE BY LINE ANALYSIS OF 2D

Lines 94 - 103 serve to separate the penultimate byte b_1 = mn in the X-register from the last byte b_0, and to store b_1 and b_0, respectively, in registers M and O. Synthetic lines 99 and 103 provide proper exponent values so that the first nybble m of b_1 (and that of b_0) will be processed by the calculator as the "integer" portion of the byte. The last four bytes of line 99 also provide the factor 0.6 (isolated in register N at line 103), which is used in decoding the bytes.

Lines 104 - 109 multiply m by 0.6 and add the result to b_1= mn, stored in register M. Line 118 multiplies register M by 10, thus completing the decoding of b_1 as 16m + n. Lines 110 - 116, 119 similarly decode b_0. Line 120 places the decoded value of b_1 into the X-register and that of b_0 into register M.

REFERENCES FOR 2D

Brief note in *PPC CALCULATOR JOURNAL*, V8N2P37a.

CONTRIBUTORS HISTORY FOR 2D

2D was conceived and written by Roger Hill (4940) as a general-purpose program pointer decoding routine for use by **LB** and **CB**, among others. **2D** saved a lot of ROM bytes. The application program "2BD" was written by Greg McCurdy (3957).

FURTHER ASSISTANCE ON 2D

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825

NOTES

TECHNICAL DETAILS

XROM: 10,55 — **2D** — SIZE: 000

Stack Usage:

- 0 T: Z
- 1 Z: Z
- 2 Y: Y
- 3 X: result 1
- 4 L: LOST

Alpha Register Usage:

- 5 M: result 2
- 6 N: USED
- 7 O: USED
- 8 P: USED

Other Status Registers: NONE USED

- 9 Q:
- 10 ├:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

ΣREG: UNCHANGED

Data Registers: NONE USED

R00:

R06:

R07:

R08:

R09:

R10:

R11:

R12:

Flag Usage: NONE USED

04:

05:

06:

07:

08:

09:

10:

25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: 1.2 seconds.

Peripherals Required: NONE

Interruptible? YES

Execute Anytime? YES

Program File: IF

Bytes In RAM: 59

Registers To Copy: 60

Other Comments:

A? - ASSIGNMENT REGISTER FINDER

If you want to know how many key assignment registers you're using, just XEQ **A?** . If you haven't already packed the key assignments you can XEQ **PK** , which packs the assignment registers and re-counts them.

A? gives a count that is either an integer or a half-integer. As long as no function assignments have been deleted, this count will be either exactly right or 1/2 register too high. **A?** gives an accurate count if all the assignments were made using **MK** , **1K** , and **+K** . The count may be 1/2 register too high if the ASN function has been used. In this case **PK** will give an accurate count.

A? counts all key assignment registers until it finds an empty one. If the right half of the top register used contains a zero keycode (last byte = 00) then the count is reduced by 1/2.

Example 1: XEQ **CK** to clear all function key assignments. XEQ **MK** and assign five functions to keys. Then XEQ **A?** and see 2.5, indicating that 2.5 key assignment registers are occupied.

Example 2: Continuing Example 1, ASN HR to any unused key. XEQ **A?** gives 3.0 a correct count. This is because ASN always fills any unused half register before opening a new one. Now ASN HMS to another unused key and XEQ **A?** . The result this time is 4.0 because ASN opened up a new register at the bottom of the key assignments, and **A?** doesn't check for a half empty register there. XEQ **PK** packs the key assignments, leaving the void in the right half of the top key assignment register and giving the correct count of 3.5 registers.

COMPLETE INSTRUCTIONS FOR A?

Just XEQ **A?** to get the assignment register count as described above. If you have deleted any assignments or used the ASN function, XEQ **PK** for an accurate count. In these cases, however, the count produced by **A?** will be at most 1/2 register off (on the high side).

Although **A?** can be interrupted or single-stepped, you should let it run to completion to avoid leaving the curtain at absolute address 16 (which would cause MEMORY LOST upon resizing). **A?** uses the alpha register and the whole stack. A temporary c register from **OM** is left in Y. Z and T are cleared, and flag 10 is cleared.

Routine Listing For: A?			
164♦LBL "A?"	167 CLA	171 .5	174 -
165 XROM "LF"	168 INT	172 FC?C 10	175 RTN
	169 175	173 SIGN	
166♦LBL 14	170 -		

LINE BY LINE ANALYSIS OF A?

Line 165 produces an ISG pointer to the free register block, relative to a curtain address of 16, with flag 10 set if the first register of that block has a key assignment in its left half. See **LF** for details. The integer part of this ISG pointer is 176 plus the number of full assignment registers. Thus lines 166-170 compute the number of assignment registers plus one, or plus 1/2 if the last register is half occupied. Lines 171-175 subtract 1/2 if flag 10 is set and 1 if flag 10 is clear.

TECHNICAL DETAILS

XROM: 10,10 — **A?** — SIZE: 000

Stack Usage:
- 0 T: 0
- 1 Z: 0
- 2 Y: temporary c
- 3 X: result
- 4 L: 1 or .5

Flag Usage: ONLY FLAG 10 IS ALTERED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10: CLEARED
- 25:

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED BUT RESTORED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 3

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R11:
- R12:

Global Labels Called:

Direct	Secondary
LF	E?
	OM
	2D

Local Labels In This Routine: 14

Execution Time: 8.7 seconds (For 16 assignment registers)

Peripherals Required: NONE

Interruptible? YES

Execute Anytime? YES

Program File: **LF**

Bytes In RAM: 22

Registers To Copy: 59

Other Comments:

CONTRIBUTORS HISTORY FOR A?

A? was written by Roger Hill (4949) as an addition to his group of key assignment programs. It is a natural extension of **LF** .

FURTHER ASSISTANCE ON A?

Call Keith Jarett (4360) at (213) 374-2583.
Call Roger Hill (4940) at (618) 656-8825.

APPENDIX A CONTINUED FROM PAGE 33.

⋮

(m-2) Transfer Q(3,e) discs to peg m-1 from peg 1.
(m-1) Transfer 1 disc to peg m from peg 1.
( ) Transfer Q(3,e) discs to peg m from peg m-1.
(m+1) Transfer Q(4,e) discs to peg m from peg m-2.

⋮

(2m-3) Transfer Q(m,e) discs to peg m from peg 2.

We see that no disc required more than $2 \cdot 2^e = 2^{e+1}$ moves. On the other hand, transferring N+1 discs would have required that one disc undergo $2 \cdot 2^{e+1} = 2^{e+2}$ moves. In other words, N = Q(m,e+1). We've established that

$$Q(m,e+1) = 1 + \Sigma\{Q(\mu,e):\mu=3,\ldots,m\}$$

But then $Q(m,e) = [1 + \Sigma\{Q(\mu,e-1): =3,\ldots,m-1\}] + Q(m,e-1)$

$$= Q(m-1,e) + Q(m,e-1)$$

A table of values for Q(m,e) will reveal the simple pattern:

e \ m	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
2	3	6	10	15	21	28	36
3	4	10	20	35	56	84	120
4	5	15	35	70	126	210	330
5	6	21	56	126	252	462	792

We now have all we need to know to solve our problem. Suppose, for example, we wish to move 25 discs using 6 pegs. This calls for partitioning the upper 24 discs into four substacks in a way which will minimize the total number of required moves. If we look at the Q(m,e) table, we see that the second row of values for m = 3, 4, 5, 6 contains 2, 3, 4, 5 totaling 14, while the third row contains 3, 6, 10, 15 totaling 34. Thus, we see that an optimum strategy requires as many as $2^3 = 8$ moves, but never more, for some discs. Any combination of four numbers n_3, n_4, n_5, n_6 such that $Q(i,1) \le n_i \le Q(i,2)$ and $\Sigma\{n_i:i=3,4,5,6\} = 24$ will suffice for a partitioning corresponding to a minimum number of moves. Note that in general there is more than one solution to a given problem. In our sample problem

$$n_3 = 2,\ n_4 = 3,\ n_5 = 10,\ n_6 = 9$$

will work as well as

$$n_3 = 3,\ n_4 = 6,\ n_5 = 10,\ n_6 = 5,$$

to mention but two of 56 possibilities.

Each of these subproblems (e.g., move $n_5 = 10$ discs using 5 pegs) can be handled similarly, until the requirement is reduced to moving a single disc.

To evaluate the number of moves required for a specific problem is straightforward: simply add up the number of moves required for each disc. Consider, for instance, our example of moving 25 discs using 6 pegs:

24-(2+3+4+5)=10 discs each requiring $2\cdot2^2$ moves -- 80
14-(1+1+1+1)=10 discs each requiring $2\cdot2^1$ moves -- 40
4 discs each requiring $2\cdot2^0$ moves -- 8
1 disc requiring 1 move -- 1

for a total of 129 moves.

As a second example, consider moving 13 discs using 5 pegs:

12-(2+3+4)=3 discs each requiring $2\cdot2^2$ moves -- 24
9-(1+1+1)=6 discs each requiring $2\cdot2^1$ moves -- 24
3 discs each requiring $2\cdot2^0$ moves -- 6
1 disc requiring 1 move -- 1

for a total of 55 moves.

The recursive routine GHT (Generalized Hanoi Tower) implements the strategy outlined for solving the m-peg version of this puzzle. Such a routine would probably be regarded as outside the scope of the HP-41 were it not for the curtain-moving and return-stack extension routines provided by the PPC Custom ROM. Naturally the memory limitations of the HP-41 impose some constraints, but cases requiring more memory than is available are also, for the most part, cases entailing too many moves for recreational interest. The data compaction schemes employed in GHT do not permit m>9 nor n>45. The number r of data registers required for legal values of m and n is given by

$$r = 9n_3 + mp + 2e + \max(6, m + 1)$$

where:

n_3 = number of discs (as evaluated by 'PARTS') to be moved to peg m-1 using only 3 pegs;

p = number of data registers allocated to each peg $= \lceil(n/5)$;

e = number of required extensions of the return stack $= \lfloor[(n_3-1)/5]$;

$\lceil z$ = least integer not less than z ('ceiling' of z);
$\lfloor z$ = greatest integer not greater than z ('floor' of z).

As long as SIZE ≥ 4, set-up routine IGT will proceed successfully, issuing prompting messages if more data registers are needed. The calling sequence is

# of pegs ↑ # of discs, XEQ 'IGT'.

If resizing prompts are displayed, resize as requested and press R/S to continue. When "READY?" is displayed, all required data for calling GHT have been established. At this point you have the option of turning on the printer to record the successive moves; press R/S to continue.

Each call on recursive routine GHT (except the first) is preceded by a call on HD to hide 9 data registers:

00 for curtain moving: set up by HD; used by UD
01 $.i_1i_2\text{---}i_{m'}$ = indices of pegs currently in use
02 n' = # of discs currently being moved
03 m' = # of pegs currently in use
04 $W.pm_0$ = global work area specification
05 subtask control: peg count
06 subtask control: peg indices

APPENDIX A CONTINUED ON PAGE 61.

AD - ALPHA DELETE LAST CHARACTER

AD is the equivalent of manually going to Alpha mode and pressing Append, then backarrow. It removes the rightmost character from the alpha register.

Example 1: Key in the alpha string "ABCDEFGHIJKLMNOPQRSTUVW", then XEQ AD to get "ABCDEFGHIJKLMNOPQRSTUV" in the alpha register.

COMPLETE INSTRUCTIONS FOR AD

Just XEQ AD to remove the rightmost character from the alpha register. Alpha may contain 0 to 24 characters, but there should be no nulls imbedded in the string. The stack is lost except for L.

APPLICATION PROGRAM 1 FOR AD

The program "ADN" listed here, deletes the rightmost n characters from alpha. Just put n in X and XEQ "ADN". "ADN" calls AD n times, keeping a count in L.

```
LBL "ADN"
SIGN
LBL 00
XROM AD
DSE L
GTO 00
END
```

Routine Listing For: AD	
98♦LBL "AD"	109 STO [
99 RCL ↑	110 RDN
100 RCL]	111 STO \
101 .	112 RDN
102 X<> \	113 STO]
103 "├*******"	114 .1
104 RCL \	115 STO ↑
105 CLA	116 ASHF
106 STO [	117 ARCL Z
107 ASTO X	118 RTN
108 RDN	

LINE BY LINE ANALYSIS OF AD

Lines 98-102 put P, O, and N in the stack. Lines 103-108 put the first 6 characters of M in the stack. Lines 109-118 reassemble the alpha register from stack contents.

CONTRIBUTORS HISTORY FOR AD

The first version of AD was written by Gerard Westen (4780). The ROM version was also written by Gerard. John McGechie (3324) has written several versions of AD which handle various numbers of characters, but this is the shortest 24-character version.

FINAL REMARKS FOR AD

Any kind of alpha processing may be able to use AD. In general, alpha handling on the 41C is slow. Iterative use of AD is also slow, though each use is fairly fast by itself.

FURTHER ASSISTANCE ON AD

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,18 — AD — SIZE: 000

Stack Usage:
- 0 T: USED
- 1 Z: USED
- 2 Y: USED
- 3 X: USED
- 4 L: UNCHANGED

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

SHIFTED ONE CHARACTER TO THE RIGHT

Other Status Registers:
- 9 Q:
- 10 ├:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

NONE USED

ΣREG: UNCHANGED

Data Registers: NONE USED

R00:

R06:

R07:

R08:

R09:

R10:

R11:

R12:

Flag Usage: NONE USED

04:

05:

06:

07:

08:

09:

10:

25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct — Secondary

NONE

Local Labels In This Routine:

NONE

Execution Time: .8 seconds

Peripherals Required: NONE

Interruptible? YES

Execute Anytime? YES

Program File: ML

Bytes In RAM: 45

Registers To Copy: 64

Other Comments:

STACK AND ALPHA REGISTER ANALYSIS FOR AD

T	Z	Y	X	L	P	O	N	M	L-#	INSTRUCTION
t	z	y	x	T	1 2 3 4 A B C	D E F G H I J	K L M N O P Q	R S T U V W X	98	♦LBL "AD"
z	y	x	P						99	RCL ↑
y	x	P	O						100	RCL]
x	P	O	Ø						101	.
			N				Cleared		102	X<> \
					C D E F G H I	J	R S T U V W	x * * * * * *	103	"⊢******"
P	O	N	"RSTUVW						104	RCL \
					Cleared	Cleared	Cleared	Cleared	105	CLA
								R S T U V W	106	STO [
			"RSTUVW"						107	ASTO X
"RSTUVW"	P	O	N						108	RDN
								K L M N O P Q	109	STO [
N	"RSTUVW"	P	O						110	RDN
							D E F G H I J		111	STO \
O	N	"RSTUVW"	P						112	RDN
						1 2 3 4 A B C			113	STO]
N	"RSTUVW"	P	.1						114	.1
					X̄ ☒ ☒				115	STO ↑
					Cleared	A B C			116	ASHF
					A B C	D E F G H I J	K L M N O P Q	R S T U V W	117	ARCL Z
N	"RSTUVW"	P	.1	T					118	RTN

AL - ALPHABETIZE X & Y

AL is a general-purpose alphabetizing subroutine. It compares two alpha strings and, if they are not already in proper alphabetical order, exchanges them.

AL may be used in either of two modes, which are selected automatically according to the nature of the contents of the X & Y registers: (1) Direct mode- If the X & Y registers contain alpha strings, XEQ AL will leave them in ascending alphabetical order in X & Y; that is, the string which is "lower" or closer to the beginning of the alphabet will be left in X, and the "higher" string in Y. (2) Indirect mode- If the X & Y registers contain numbers, AL will interpret them as indirect addresses and will alphabetize the character strings in the two data registers addressed by X & Y. If the register addresses in X & Y are in ascending order, the strings will be alphabetized in ascending order; if the register addresses in X & Y are in descending order, the strings will be alphabetized in descending order.

Example 1: Direct mode. (d= don't care, g= garbage)

```
T      d                       T      g
Z      d                       Z      g
Y      "ALPHA"    XEQ AL →     Y      "BETA"
X      "BETA"                  X      "ALPHA"
L      d                       L      g
ALPHA  d                       ALPHA  [clear]

T      d                       T      g
Z      d                       Z      g
Y      "BETA"     XEQ AL →     Y      "BETA"
X      "ALPHA"                 X      "ALPHA"
L      d                       L      g
ALPHA  d                       ALPHA  [clear]
```

Example 2: Indirect mode (ascending).

```
T      d                       T      90
Z      d                       Z      89
Y      90         XEQ AL →     Y      g
X      89                      X      g
L      d                       L      g
ALPHA  d                       ALPHA  [clear]
R89    "BETA"                  R89    "ALPHA"
R90    "ALPHA"                 R90    "BETA"
```

Example 3: Indirect mode (descending).

```
T      d                       T      89
Z      d                       Z      90
Y      89         XEQ AL →     Y      g
X      90                      X      g
L      d                       L      g
ALPHA  d                       ALPHA  [clear]
R89    "ALPHA"                 R89    "BETA"
R90    "BETA"                  R90    "ALPHA"
```

COMPLETE INSTRUCTIONS FOR AL

The alpha strings on which AL operates may be of different lengths, up to the maximum of six characters which can be held in a data register as a result of the ASTO command. For example, "AAAA" will be placed ahead of "AAAAAA". Any character in the 41C's character set (including those from the lower half of the combined hex table) can be included, and they will be "alphabetized" in the order of their decimal or hex numbers as set forth in the combined hex table. For example, "3BB" will be placed ahead of "4AA". In terms of printable characters, the strings will be alphabetized in the order of their "BLDSPEC" numbers. One unfortunate consequence to remember (common to all systems using ASCII codes) is that the entire set of lower case letters comes after the entire set of upper case letters, so that "alpha" will be placed after "BETA".

Stack usage is shown in the Examples, above.

APPLICATION PROGRAM 1 FOR AL

The routine ACMP listed below is a faster alphabetizer that works only in the indirect mode. In that mode it operates identically to AL. The AORD routine below accepts an ISG/DSE pointer of the form bbb.eeeii and uses ACMP to alpha-sort the contents of the chosen block of registers in ascending order. Both routines were written by Roger Hill (4940). If you need more speed than AL and you don't need the direct mode, or if you want to sort alpha data, use these routines.

BAR CODE ON PAGE 484

APPLICATION PROGRAM FOR: AL

```
01•LBL "AORD"
02 CF 10
03 ENTER↑
04 ENTER↑
05 FRC
06 ST- Y
07  E-3
08 ST- T
09 ST* Z
10 /
11 +

12•LBL 05
13 ABS
14 X<>Y

15•LBL 06
16 XEQ "ACMP"
17 R↑
18 R↑
19 DSE Y
20 GTO 06
21 LASTX
22  E-3
23 +
24 ISG Y
25 GTO 05
26 RTN

27•LBL "ACMP"
28 SF 09

29•LBL 01
30 "*x"
31 ARCL IND Y
32 "├••••"
33 ASTO [
34 "├••"
35 RCL [
36 FS?C 09
37 GTO 01
38 X<Y?
39 RTN
40 X=Y?
41 GTO 03

42•LBL 02
43 FS?C 10
44 RTN
45 X<> IND Z
46 X<> IND T
47 X<> IND Z
48 RTN
49•LBL 03
50 R↑
51 R↑
52 SF 09

53•LBL 04
54 "**"
55 ARCL IND Y
56 ASHF
57 ASTO T
58 "x"
59 ARCL T
60 "├•••••"
61 ASTO [
62 "├•"
63 RCL [
64 FS?C 09
65 GTO 04
66 X>Y?
67 GTO 02
68 .END.
```

LINE BY LINE ANALYSIS OF AL

The byte manipulation undertaken by the routine at LBL 14 left-justifies shorter strings in the register, else leading null bytes would result in shorter strings being alphabetized ahead of longer strings even if their first character was higher. The left-justification is cleverly achieved by lines 158-160. The alpha register was first cleared to nulls at line 121. Line 158 then pushes even a one-character string into the sixth position from the right, by appending five nulls. ASTO L then stores six alpha bytes into the L register, beginning with the left-most character in the alpha register -- that is, with the first character of the original string. In the case of a one-character string, for example, ASTO L places that one character followed by the five appended null bytes into the L register. Finally, ARCL L appends the resulting six-character string back onto the right end of the alpha register, whence it is pushed five places to the left by line 161.

The second LBL 14 at line 167 then obliterates the trailing two bytes of the string by storing 0 in the M register, pushes the string the rest of the way into N, and recalls it into X. At this point, the first four

Routine Listing For: AL

```
120*LBL "AL"
121 CLA
122 CF 10
123 XEQ 14
124 XEQ 14
125 X=Y?
126 XEQ 13
127 CLA
128 SF 10
129 X<=Y?
130 CF 10
131 FC?C 25
132 GTO 12
133 X<=Y?
134 RTN
135 X<> IND T
136 X<> IND Z
137 X<> IND T
138 RTN

139*LBL 12
140 X<=Y?
141 GTO 12
142 R↑
143 X<> Z

144*LBL 12
145 R↑
146 R↑
147 RTN

148*LBL 13
149 R↑
150 R↑
151 SF 10
152 XEQ 14

153*LBL 14
154 SF 25
155 ARCL IND Y
156 FC? 25
157 ARCL Y
158 "├••••••"
159 ASTO L
160 ARCL L
161 "├••••••"
162 .
163 FC? 10
164 GTO 14
165 STO \
166 "├••"

167*LBL 14
168 STO [
169 "├••"
170 X<> \
171 RTN
```

characters of the string occupy the second through fifth bytes in X. Because the first, sixth and seventh positions are null, the 41C interprets the string in X as a positive number, +0.nnnnnnnn0 * 10 to the 00 power. The X<Y and X>Y tests can operate on such a string.

The original contents of X have now been pushed up to Y, and they are processed in the same manner by a second call of LBL 14.

Once the two four-character strings are in X and Y, they are compared at line 125. If they are not equal, **AL** is able to alphabetize the original strings without more information. The test of flag 25 at line 131 sends the routine to LBL 12, to place the original strings in the proper order in X and Y if **AL** is in direct mode, or to place the original strings in the proper order in the original data registers if it is in indirect mode. In either case, the original strings had been saved in the stack for this purpose.

If the test at line 125 reveals that the first four characters of the two strings are identical (e.g., "CCCCAX" and "CCCCB"), then more information is needed. **AL** therefore branches to LBL 13, executes LBL 14 on the last two characters of each string, and returns to line 127 for final alphabetization.

CONTRIBUTORS HISTORY FOR AL

First version written by Bill Wickes (3735) and published in *PPC CJ*, V7N3P7. Versions closer to the present listing were published in W.C. Wickes, *Synthetic Programming on the HP-41C* (Larken Publications, 1980) p. 67, and by Keith Jarrett (4360) in *PPC CJ*, V7N7P19. The original Wickes and the Jarrett versions contained (different ?)bugs which could generate errors in the case of strings longer than 4 characters (see *PPC CJ*, V7N6P6, V6N10P16). William Cheeseman revised **AL** to work correctly on all strings of six or fewer characters, and he added the ability to select automatically between direct and indirect mode.

FINAL REMARKS FOR AL

AL is not fast, but it is faster in the general case than a routine which compares one character at a time until a difference is found, because **AL** always compares the first four characters at once (and the last two characters at once, if necessary).

The most obvious application of **AL** is to use it as the core of a program to alphabetize a long list in a block of data registers. See *PPC CJ*, V7N9P18 regarding the TI club's challenge to find the fastest routine to sort 99 five-letter strings. Because **AL** can exchange adjacent string pairs, a Bubble Sort is an obvious technique. A faster algorithm may be possible, however, by taking advantage of the three facts that (1) the indirect mode pointers need not point to adjacent registers, (2) the pointers survive in Z and T, and (3) flag 10 is left set if an exchange took place, clear if it did not.

FURTHER ASSISTANCE ON AL

Call William Cheeseman (4381) at (617) 235-8863.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,37	AL	SIZE: 000

Stack Usage: (DIRECT MODE)			Flag Usage: ONLY FLAGS 10 AND 25 USED	
0 T:	USED		04:	
1 Z:	USED		05:	
2 Y:	Y	or X	06:	
3 X:	X	or Y	07:	
4 L:			08:	
Alpha Register Usage:			09:	
5 M:			10:	SET IF EXCHANGED CLEARED IF NOT
6 N:	ALL CLEARED			
7 O:				
8 P:			25:	CLEARED

Other Status Registers:		Display Mode: UNCHANGED
9 Q:		
10 ├:	NONE USED	
11 a:		Angular Mode: UNCHANGED
12 b:		
13 c:		
14 d:		Unused Subroutine Levels:
15 e:		4

ΣREG: UNCHANGED	Global Labels Called:	
Data Registers: NONE USED	Direct	Secondary
R00:	NONE	NONE
R11:	Local Labels In This Routine:	
R12:	12 TWICE 13 14 TWICE	

Execution Time: 1.6 - 2.6 seconds.

Peripherals Required: NONE

		Other Comments:
Interruptible?	YES	For INDIRECT MODE X and Y are lifted to Z and T.
Execute Anytime?	NO	
Program File:	VK	
Bytes In RAM:	105	
Registers To Copy:	63	

AM - ALPHA TO MEMORY

The **AM** routine is used to ALPHA STORE the contents of the ALPHA register (or a part of it) into four data registers. The ALPHA register is cleared when **AM** is finished. The inverse routine, **MA**, is used to restore the ALPHA register by ALPHA RECALLING, in order, the four data registers used by **AM**. Both routines require the standard "ISG" format of bbb.eeeii as the only input. The normal input is bbb.eee with the ii part being the default value of 00 for an increment value of 1.

AM and **MA** were placed into the PPC ROM as last minute "byte fillers". See **PO** for additional information on why "byte fillers" were used.

Example 1: Store the ALPHA register into data registers 1 thru 4. Fill ALPHA with: ABCDEFGHIJKLMNOPQRSTUVWX and store it as shown below.

DO:	SEE:	RESULT:
1.004	1.004	ISG input for **AM**
XEQ **AM**	5.0040	A-X in ALPHA now in R1-R4
RCL 01	ABCDEF	Left six ALPHA characters
RCL 02	GHIJKL	Second six ALPHA characters
RCL 03	MNOPQR	Third six ALPHA characters
RCL 04	STUVWX	Right six ALPHA characters

Example 2: Store the ALPHA register into data registers 12, 14, 16, & 18. Fill ALPHA with ABCDEFGHIJKLMNOPQRSTUVWX and store it as shown below.

DO:	SEE:	RESULT:
XEQ "CLRG"	No Change	Data registers cleared
12.01802	12.01802	ISG input for **AM**
XEQ **AM**	20.01802	A-X in ALPHA now in 12, 14, 16, & 18
RCL 12	ABCDEF	
RCL 14	GHIJKL	As expected. R13, 15, & 17 are ZERO.
RCL 16	MNOPQR	
RCL 18	STUVWX	

COMPLETE INSTRUCTIONS FOR AM

A bbb.eeeii "ISG" control number is required for **AM** to work properly. The limited number of bytes available for this routine did not allow for X register "clean-up", reprocessing the "ISG" control number, or providing for a specific register to act as an ISG counter. The bbb.eeeii control may be used effectively to store and clear 1/4, 1/2, 3/4, or all of the ALPHA register. The data registers used to store the contents of the ALPHA register may be "selected" within the limits of the ii increment. Typical examples of inputs suitable for **AM** (and **MA**) are:

A -	1.004	F -	5.02505
B -	.003	G -	1.002
C -	.01	H -	.001 (E-3)
D -	12.015	I -	28.03
E -	6.01202	J -	2.01311

A. ALPHA register contents stored in R1-R4.
B. ALPHA register contents stored in R0-R3.
C. Minimum bytes, stores ALPHA register contents in R0-R3 and six NULLS in R4-R10.
D. ALPHA register contents stored in R12-R15.
E. ALPHA register contents stored in R6, R8, R10, R12.
F. ALPHA register contents stored in R5, R10, R15, R20.
G. Half of ALPHA register stored in R1 & R2. Second half remains unchanged in ALPHA.
H. Half of ALPHA register stored in R0 & R1. Use of E-3 saves a byte.
I. 3/4 of ALPHA register stored in R28, 29, 30.
J. Half of ALPHA register stored in R2, & R13.

AM uses only the X register and may be used to store the ALPHA register to preserve it for future use. The same control numbers are used for the inverse, **MA**, routine.

MORE EXAMPLES OF AM

Example 3: Write a routine to store the ALPHA register in R0-R3 and resume execution with ALPHA still having its original contents.

```
01  0.003        04  LAST X
02  ABS          05  XROM MA
03  XROM AM      06  Continue
```

LINE BY LINE ANALYSIS OF AM

LINE NO.

- 37. Global label.
- 38. Local label for loop entry.
- 39. ALPHA store six characters into register of integer value of X register.
- 40. SHIFT ALPHA register six characters.
- 41. Increment the X register, branch out of loop if eee value is reached.
- 42. Repeat loop going to LBL 01.
- 43. Return to RAM calling program.

Routine Listing For: AM

```
37♦LBL "AM"        41 ISG X
38♦LBL 01          42 GTO 01
39 ASTO IND X      43 RTN
40 ASHF
```

CONTRIBUTORS HISTORY FOR AM

AM was conceived by Keith Jarett (4360) and Richard Nelson (1) during an early morning SDS loading session.

T E C H N I C A L   D E T A I L S

XROM: 20,53 | **AM** | SIZE: ≥ 005

Stack Usage:
- 0 T: NOT USED
- 1 Z: NOT USED
- 2 Y: NOT USED
- 3 X: INPUT ISG COUNTER
- 4 L: NOT USED

Alpha Register Usage:
- 5 M:
- 6 N: Cleared after
- 7 O: use.
- 8 P:

Other Status Registers:
- 9 Q:
- 10 ├:
- 11 a:
- 12 b:
- 13 c: NONE USED
- 14 d:
- 15 e:

ΣREG: NOT USED

Data Registers:

R00: NOT USED

R06:
R07: 1 to 4 Registers
R08: selected by user.
R09:
R10:
R11:
R12:

Flag Usage: NONE

04:
05:
06:
07:
08:
09:
10:

25:

Display Mode: N/A

Angular Mode: N/A

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
NONE	

Local Labels In This Routine:

LBL 01

Execution Time: 1 second.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	NS
Bytes In RAM:	17
Registers To Copy:	16

Other Comments:

FINAL REMARKS FOR AM

AM, and it's inverse, MA, are routine functions of the "block" type. Improvements could be made by having R06 or other register used for the control number. Upon completion the control number should be restored to it's start value. Perhaps the only input should be the start register value and all four alpha registers are stored. AM is not a synthetic register store routine, i.e., M, N, O, P. A future routine may perform this task.

FURTHER ASSISTANCE ON AM

Call Richard Nelson (1) at (714) 754-6226 (P.M.).
Call Richard Schwartz (2289) at (213) 447-6574.

NOTES

Ab - ALPHA STORE b

Ab and Sb are both one-instruction "programs" that provide an ultra-fast ROM entry capability as an alternative to XE (XROM entry). Ab consists of an ASTO b instruction which stores a user-specified code in the program pointer, immediately transferring execution to another point in ROM. XE can be thought of as an XEQ IND function. Similarly, Ab and Sb can be thought of as GTO IND functions. XE , Ab , and Sb use the contents of an indirect register (ALPHA or X) as an entry address. XE preserves up to five subroutine returns including the one to the calling RAM program. On the other hand, Ab and Sb destroy all pending subroutine returns and program execution ultimately halts in ROM, unless synthetically constructed returns were provided.

COMPLETE INSTRUCTIONS FOR Ab

At first glance, Ab and Sb don't seem to offer any byte savings in the user's RAM programs. XROM Ab and XROM Sb each require two bytes of RAM which is the same as ASTO b and STO b require; therefore, the Ab and Sb labels appear to be taking up space in the PPC ROM without purpose. However, this is not the case as we shall see.

For example, do the following:

0) Assign RCL b and STO b to keys (use MK).
1) GTO "PRAXIS". (a printer ROM routine)
2) RCL b (or ARCL b).
3) XEQ CATALOG 1.
4) STO b (or ASTO b).
5) PRGM mode on (Note: program pointer is <u>not</u> at "PRAXIS").
6) PRGM mode off.
7) GTO "PRPLOT" (or any ROM global other than "PRAXIS").
8) STO b (or ASTO b).
9) PRGM mode on (Note: program pointer <u>is</u> at "PRAXIS").
10) PRGM mode off.

Why did steps 7 to 9 produce the desired result (i.e, locate "PRAXIS") while steps 3 to 5 did not?

The GTO "PRAXIS" set the program pointer (in register b) at ROM address 6108 (i.e., byte 108_{16} of ROM 6 - the printer). The RCL b brought this value (6108) to the X register. Execution of CATALOG 1 reset the program pointer to a RAM address (in a user program). When in RAM, the address 6108 is interpreted as byte 6 of register 108_{16}. Thus, the subsequent STO b sets the program pointer to this location in RAM (if it exists and is occupied) and not back to the location of "PRAXIS".

However, GTO "PRPLOT" sets the program pointer to a ROM address, so that when another STO b is executed, 6108 is once again interpreted as a ROM address and the program pointer is reset to "PRAXIS".

PPC ROM routines Ab and Sb automate the manual process given in the above example. If the ALPHA or X register contains a code which represents an address (e.g., 6108), then XROM Ab or XROM Sb will cause this address to be interpreted as a ROM address. When an user's calling program encounters XROM Ab or XROM Sb , program execution is transferred to the PPC ROM. Since the program pointer now represents a ROM address (in the PPC ROM), the subsequent ASTO b or STO b (in Ab or Sb) will cause the running program to jump to the ROM address (e.g., 6108) which the user provided in the ALPHA or X register. Program execution will continue until an END, RTN or STOP is encountered in the ROM program, at which time a halt will occur with the program pointer in ROM.

Note that in the above example, program execution jumped from a user's RAM program to the PPC ROM and then to the printer ROM. This all occurred as the result of XROM Ab or XROM Sb , two short but powerful instructions.

Ab and Sb destroy all subroutine returns that are pending at the time Ab or Sb is executed. If the pending returns are needed, then the PPC ROM routine XE should be used instead of Ab or Sb .

<u>Example 1</u>: The following routine demonstrates the use of Ab (or Sb) in a RAM program. This program has these features:

1) It is named "VK" which is the name of the PPC ROM routine which is to be enhanced/modified. The RAM "VK" rather than the ROM VK will be called by an XEQ "VK", because CATALOG 1 global labels are found before CATALOG 2 global labels or functions during a search.

2) It adds Roger Hill's (4940) original "KEYS USED:" as a replacement for the stalled "flying goose".

3) It bypasses Lines 01-07 of the PPC ROM routine VK which allows "VK" to operate with the printer present, but turned <u>off</u>. This routine will not work with the printer present and turned on.

<u>USING Ab :</u>

```
01  LBL "VK"
02  CF 21
03  "KEYS USED:"
04  AVIEW
05  hex F2 W9 33
06  XROM Ab
```

<u>USING Sb :</u>

```
01  LBL "VK"
02  CF 21
03  "KEYS USED:"
04  AVIEW
05  hex F2 W9 33
06  RCL M
07  XROM Sb
```

Line 05 is a two byte text line which represents the address in ROM to which the GTO IND is to jump. The rightmost three hexadecimal digits (933 in this case) give the location of the jump destination within a 4K ROM. (Note that there are 4096 possible values for three hex digits). The remaining, or leftmost, hexadecimal digit indicated by W here, gives the port address of the 4K ROM as follows: 8 = Port 1 lower 4K, 9 = Port 1 Upper 4K, A = Port 2L, B = Port 2U, C = Port 3L, D = Port 3U, E = Port 4L, F = Port 4U. Since VK is in the lower 4K of the PPC ROM (the lower 4K appears first in CATalog 2), W will be 8, A, C, or E here according as the PPC ROM is in port 1, 2, 3, or 4. Line 05 can be created using LB or the text Q-loader (see *PPC CALCULATOR JOURNAL*, V7N8P27a).

In the "VK" examples above, the use of Ab is preferred to Sb because it saves bytes. In other applications for which the NNN is already in the X register, Sb might be preferred. Both examples above transfer program execution to line 08 of PPC ROM routine VK and the program ultimately halts in the PPC ROM.

<u>Example 2</u>: The two byte text line which is created for the above examples makes the RAM "VK" routine port-dependent; that is, the routine will work correctly only if the PPC ROM is installed in the port represented by W. This can be remedied by the PPC ROM routine Rb . The following routine calls a RAM routine "PE" (PPC ROM ENTRY) which in turn calls PPC

ROM routine Rb. Rb recalls the contents of register b which contains the address (including the port number) of that point in the PPC ROM. "PE" modifies the two-byte pointer which is provided by the calling program (the RAM "VK" routine in this case) to give it the port number of the PPC ROM which was obtained by Rb. The pointer supplied to "PE" should be either 8 ijk or 9 ijk depending upon which half of the PPC ROM is to be entered. In effect the calling RAM program addresses Port 1 and the "PE"/ Rb combination modifies the call to address the correct port. Thus, the calling RAM program becomes port-independent.

USER PROGRAM:

```
01  LBL "VK"            04  Hex F2 89 33
02  CF 21               05  XEQ "PE"
03  "KEYS USED:"        06  XROM Sb
```

PPC ROM ENTRY SETUP PROGRAM:

```
01  LBL "PE"               12  FC? 01    25  FRC
02  XROM Rb                13  CLX       26  X≠0?
03  X<>M                   14  +         27  SF 02
04  Hex F6 7F 00 00 00     15  X<>Y      28  X<>M
    00 00                  16  X<>M      29  X<>d
05  X<>M                   17  X<>d      30  X<>M
06  Hex F6 7F 00 00 00     18  RDN       31  Hex F3 7F 00 00
    00 00                  19  2         32  X<>N
07  X<>d                   20  /         33  CLA
08  .                      21  INT       34  END
09  FS? 02                 22  X≠0?
10  E↑X                    23  SF 01
11  2                      24  LASTX
```

LINE BY LINE ANALYSIS OF Ab

See the Complete Instructions above for the theory behind the operation of Ab and Sb.

CONTRIBUTORS HISTORY FOR Ab

Ab and Sb owe their existence to Tom Cadwallader (3502). Tom requested their inclusion after Bill Pickard (3514) discovered that STO b behaved differently when the program pointer was already in ROM. Bill had been trying to get into internal ROM 0 using Charles Close's (3878) "ROM + " program (see *PPC CALCULATOR JOURNAL*, V8N1P14). The STO b behavior described here was also discovered independently by Robert Groom (5127). The "PE" application routine was written by Tom Cadwallader (3502).

FINAL REMARKS FOR Ab

The HP-41C's MPU apparently has some means (Flag ?) of knowing whether the ROM instructions that it is executing are user language or assembly language. When we learn how to make the MPU recognize that ROM contents at a given point are assembly language, we can then use Ab and Sb to begin execution at that point.

Routine Listing For:	Ab
181♦LBL "Ab" 182 ASTO b	

FURTHER ASSISTANCE ON Ab

Call Tom Cadwallader (3502) at (406) 727-6869.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,61 — **Ab** — SIZE: 000

Stack Usage:
0 T:
1 Z:
2 Y:
3 X:
4 L:
ALL UNCHANGED

Alpha Register Usage:
5 M:
6 N:
7 O:
8 P:
ALL UNCHANGED

Other Status Registers:
9 Q: NOT USED
10 ⊦: NOT USED
11 a: NOT USED
12 b: ALTERED
13 c: NOT USED
14 d: NOT USED
15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 0

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine:
NONE

Execution Time: Less than .1 second.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: IF
Bytes In RAM: 8
Registers To Copy: 60

Other Comments: New contents of b is interpreted as a ROM pointer.

BA - BARCODE ANALYZER

This program analyzes single lines of HP41C barcode for type and other information on content. The display prompts for wand scanning, after which the content of the barcode is printer on the 82143A printer, which is required for this program.

Example 1. Analyze the following program barcode line with routine BA :

```
TYPE 1 PC
BY#    BIN     DEC HEX CH
 1  11110100  244  F4
 2  00010000   16  10  θ
 3  00000001    1  01  ×
 4  11000000  192  C0
 5  00000000    0  00  ♦
 6  11110101  245  F5
 7  00000000    0  00  ♦
 8  01010100   84  54  T
 9  01000101   69  45  E
10  01010011   83  53  S
11  00110100   52  34  4
12  10001110  142  8E
13  10001100  140  8C
14  10010000  144  90
15  10101101  173  AD
16  10010010  146  92
 CHECKSUM:
    11010100  212  D4
```

Figure 1. Printed output of the BA routine from analyzing the barcode from example 1.

Note that in the output of Example 1, the abbreviation 'PC' was printed, denoting that the barcode row is program code (unprivate). A complete list of the abbreviations for all barcode types appears in table 1.

ABBREV.	TYPE #	BARCODE TYPE
PC	1	Program, unprivate
PP	2	Program, private
DE	4	Direct Execution
PK	5	Paper Keyboard
ND	6	Numeric Data
AR	7	ALPHA Replace Data
AA	8	ALPHA Append Data
NS	9	Numeric Sequenced Data
ARS	10	ALPHA Replace Sequenced Data
AAS	11	ALPHA Append Sequenced Data
UT	0,3,12-15	Unused Type

Table 1. List of abbreviations for barcode types as they would appear as output of the BA program. All other types would show type numbers if scanned followed by UT for unused type.

COMPLETE INSTRUCTIONS FOR BA

The routine is initiated by pressing XEQ BA. The display then prompts 'SCAN' and the user scans the row of barcode. Then the barcode type number (between 0 and 15 inclusive) and abbreviated name is printed. If the type is nonstandard, the type number is printed but no type abbreviation accompanies it. The printer then prints the information on the individual barcode bytes in binary, decimal, hexadecimal and equivalent printer character if the byte if less than or equal to 7F (hex) or 127 (decimal). After the individual byte information, the 8-bit checksum, computed from the sum of the 2nd through last barcode bytes, is printed. This may be compared to checksum byte (#1) in the row of barcode itself, for correctness. This computed checksum is printed only if the barcode is determined to not be paper keyboard type. The routine calls a barcode paper keyboard type if it is 1 or 2 bytes in length. Since the only legitimate barcode of this length is the paper keyboard type, BA automatically assumes these short codes to be of that type. This means that other nonstandard codes which are also type 5 (the internally set type of the paper keyboard code) and are longer than 2 bytes may be scanned and analyzed.

If flag 10 is clear, then the program prompts the user to scan another row after the analysis of the first row is complete. If F10 is set, then the program stops.

FURTHER DISCUSSION OF BA

Limitations:

This routine uses WNDSCN to read barcodes. The WNDSCN format does not compare a calculated checksum against the first byte of the barcode row, therefore erroneous scans may go unnoticed by the wand. The only safeguard in WNDSCN mode is against accepting rows which do not have an even multiple of 8 bars (disregarding the 2 start and 2 stop bars at the ends). Therefore, if calculated checksums do not agree with the value of byte number 1 in the row, then the possibility exists that the row was scanned unreliably, in addition to the possibility that the actual checksum byte is incorrect. If the checksum does not agree with byte #1, it is therefore recommended that the barcode row be scanned again. One should also note that the checksum in program (types 1 or 2) barcode rows beyond row number one will contain the running checksum for all rows, rather than the sum for the scanned row alone. Do not expect the calculated checksums from these rows to be the same as that in byte #1.

MORE EXAMPLES OF **BA**

Example 2. Eleven barcodes of varying types appear below. Analyze them with the **BA** routine.

TYPE 2 PP

BY#	BIN	DEC	HEX	CH
1	10101111	175	AF	
2	00100001	33	21	!
3	00000001	1	01	x
4	10001111	143	8F	
5	11110101	245	F5	
6	01111111	127	7F	⊢
7	01000011	67	43	C
8	01001100	76	4C	L
9	01010101	85	55	U
10	01000010	66	42	B
11	00100101	37	25	%
12	01011001	89	59	Y
13	01011110	94	5E	↑
14	01110110	118	76	v
15	10001110	142	8E	
16	11000000	192	C0	
CHECKSUM:				
	11110000	240	F0	

PROMPT

SIN

TYPE 4 DE

BY#	BIN	DEC	HEX	CH
1	11001110	206	CE	
2	01000000	64	40	@
3	10001110	142	8E	
CHECKSUM:				
	11001110	206	CE	

TYPE 5 PK

BY#	BIN	DEC	HEX	CH
1	11100000	224	E0	
2	01011001	89	59	Y

DATA 2.45E-06

TYPE 6 ND

BY#	BIN	DEC	HEX	CH
1	11001110	206	CE	
2	01101010	106	6A	j
3	00101011	43	2B	+
4	01000101	69	45	E
5	11101101	237	ED	
6	00000110	6	06	Γ
CHECKSUM:				
	11001110	206	CE	

DATA 'MICRO'

TYPE 7 AR

BY#	BIN	DEC	HEX	CH
1	11110000	240	F0	
2	01110101	117	75	u
3	01001101	77	4D	M
4	01001001	73	49	I
5	01000011	67	43	C
6	01010010	82	52	R
7	01001111	79	4F	O
CHECKSUM:				
	11110000	240	F0	

DATA A'PPC ROM'

TYPE 8 AA

BY#	BIN	DEC	HEX	CH
1	01111010	122	7A	z
2	10000111	135	87	
3	01010000	80	50	P
4	01010000	80	50	P
5	01000011	67	43	C
6	00100000	32	20	
7	01010010	82	52	R
8	01001111	79	4F	O
9	01001101	77	4D	M
CHECKSUM:				
	01111010	122	7A	

DATASEQ -2.78E-78 02

TYPE 9 NS

BY#	BIN	DEC	HEX	CH
1	01001001	73	49	I
2	10010000	144	90	
3	00000001	1	01	x
4	10101101	173	AD	
5	00101011	43	2B	+
6	01111000	120	78	x
7	11101101	237	ED	
8	01111000	120	78	x
CHECKSUM:				
	01001001	73	49	

DATASEQ 'STORE' 01

TYPE 10 ARS

BY#	BIN	DEC	HEX	CH
1	00101111	47	2F	/
2	10100000	160	A0	
3	00000000	0	00	♦
4	01010011	83	53	S
5	01010100	84	54	T
6	01001111	79	4F	O
7	01010010	82	52	R
8	01000101	69	45	E
CHECKSUM:	00101111	47	2F	

4

TYPE 5 PK

BY#	BIN	DEC	HEX	CH
1	00100100	36	24	$

WALL

TYPE 5 PK

BY#	BIN	DEC	HEX	CH
1	11101000	232	E8	
2	10001110	142	8E	

DATASEQ A'EPROM BOX' 05

TYPE 11 AAS

BY#	BIN	DEC	HEX	CH
1	01000011	67	43	C
2	10110000	176	B0	
3	00000100	4	04	α
4	01000101	69	45	E
5	01010000	80	50	P
6	01010010	82	52	R
7	01001111	79	4F	O
8	01001101	77	4D	M
9	00100000	32	20	
10	01000010	66	42	B
11	01001111	79	4F	O
12	01011000	88	58	X
CHECKSUM:	01000011	67	43	

Routine Listing For: **BA**

```
01♦LBL "BA"
02♦LBL 08
03 "SCAN"
04 CF 21
05 AVIEW
06 WNDSCN
07 SF 21
08 STO 18
09 FIX 0
10 2
11 X<Y?
12 GTO 00
13 5
14 SF 10
15 GTO 01
16♦LBL 00
17 RCL 02
18 16
19 /
20 INT
21♦LBL 01
22 .
23 X<> d
24 SF IND Y
25 "UT"
26 FS?C 01
27 "PC"
28 FS?C 02
29 "PP"
30 FS?C 04
31 "DE"
32 FS?C 05
33 "PK"
34 FS?C 06
35 "ND"
36 FS?C 07
37 "AR"
38 FS?C 08
39 "AA"
40 FS?C 09
41 "NS"
42 FS?C 10
43 "ARS"
44 FS?C 11
45 "AAS"
46 X<> d
47 ASTO Z
48 "TYPE "
49 ARCL Y
50 "⊦ "
51 ARCL Z
52 PRA
53 RCL 18
54 CF 29
55 E3
56 /
57 ISG X
58 STO 17
59 CF 12
60 "BY#    BIN"
61 "⊦    DEC HEX CH"
62 PRA
63 CF 09
64 CF 08
65 RCL 01
66 CHS
67 STO 00
68♦LBL 02
69 CLA
70 ARCL 17
71 "⊦  "
72♦LBL 03
73 8
74 RCL IND 17
75 ENTER↑
76 FC? 09
77 ST+ 00
78 128
79 X<=Y?
80 ST- Z
81 X<=Y?
82 SF 08
83 /
84♦LBL 04
85 ENTER↑
86 INT
87 ARCL X
88 -
89 ST+ X
90 DSE Z
91 GTO 04
92 RDN
93 RCL IND 17
94 "⊦  "
95 E1
96 X>Y?
97 "⊦ "
98 X↑2
99 X>Y?
100 "⊦ "
101 ARCL Y
102 "⊦  "
103 %
104 ACA
105 16
106 MOD
107 X<>Y
108 LASTX
109 /
110 INT
111 XEQ 06
112 X<>Y
113 XEQ 06
114 FS?C 09
115 GTO 05
116 3
117 FC? 08
118 DSE X
119 SKPCHR
120 R↑
121 FC?C 08
122 ACCHR
123 ADV
124 ISG 17
125 GTO 02
126 FS?C 10
127 GTO 05
128 " CHECKSUM:"
129 PRA
130 CLA
131 CLX
132 STO 17
133 SF 09
134 RCL 00
135 X=0?
136 GTO 03
137 255
138 MOD
139 X=0?
140 LASTX
141 STO 00
142 GTO 03
143♦LBL 05
144 3
145 SKPCHR
146 ADV
147 FS?C 07
148 RTN
149 GTO 08
150♦LBL 06
151 9
152 X<Y?
153 GTO 07
154 RDN
155 CLA
156 ARCL X
157 ACA
```

TECHNICAL DETAILS

XROM: 20,30	BA	SIZE: 019

Stack Usage: 0 T: 1 Z: All USED 2 Y: 3 X: 4 L:	**Flag Usage:** 04: NOT USED 05: NOT USED 06: NOT USED 07: USED 08: USED 09: USED 10: USED 12: USED 21: USED 25: NOT USED
Alpha Register Usage: 5 M: 6 N: ALL USED 7 O: 8 P:	
Other Status Registers: 9 Q: NOT USED 10 ⊢: NOT USED 11 a: NOT USED 12 b: NOT USED 13 c: NOT USED 14 d: USED 15 e: NOT USED	**Display Mode:** ANY Routine sets display mode internally. **Angular Mode:** ANY **Unused Subroutine Levels:** 4
ΣREG: NOT USED **Data Registers:** R00: USED R01 to R05: USED R06: USED R07: USED R08: USED R09: USED R10: USED R11: USED R12: USED R13 to R18: USED	**Global Labels Called:** Direct: NONE Secondary: NONE **Local Labels In This Routine:** 00 to 08
Execution Time: 3 + 5* (#bytes scanned) seconds.	
Peripherals Required: 82153A Wand, 82143A Printer	
Interruptible? YES Execute Anytime? YES Program File: BA Bytes In RAM: 337 Registers To Copy: 49	**Other Comments:** This routine saves flag register d while using flags 01 to 11 for testing the barcode type; then restores it. Only 1 port is unoccupied while running BA (PPC ROM, Wand & Printer in 3 ports).

```
158 RTN        162 +
159♦LBL 07     163 ACCHR
160 RDN        164 END
161 55
```

LINE BY LINE ANALYSIS OF BA

Lines 02 to 06 prompt the user to scan the barcode row.

Lines 07 to 12 test for barcode rows of 3 bytes or longer.

Lines 13 to 15 set type 5 (paper keyboard type) if the scanned row is less than 3 bytes in length.

Lines 16 to 20 extract the type nybble from the second barcode byte and store for later access.

Lines 21 through 52 test barcode type by setting flags, saving the later restoring the original flag register.

Lines 53 to 62 set up the printer output format for the barcode byte information.

Lines 63 to 126 analyze the individual barcode bytes for binary, hex and decimal content.

Lines 127 to 142 calculate the checksum from bytes 2 through the end of the barcode row.

Lines 143 to 158 finish off housekeeping chores associated with analysis of each barcode byte.

Lines 159 to 164 produce the ACCHR printer character from the decimal value of the barcode byte.

REFERENCES FOR BA

See PPC Calculator Journal, V7N5P30,31.

CONTRIBUTORS HISTORY FOR BA

This program represents the union of the Program to Analyze Program Barcode by Richard Nelson (1) and the Barcode Type Analyzer by Jake Schwartz (1820). The final version was achieved through the assistance of David Spear (5488) and Roger Hill (4940).

FURTHER ASSISTANCE ON BA

Contact Jake Schwartz (1820) at 7700 Fairfield St., Phila., Pa. 19152, phone 215-331-5324 evenings; or Roger Hill (4940) at 300 S. Main St., Apt 5, Edwardsville, Ill. 62025, phone 618-656-8825 evenings.

BC - BLOCK CLEAR

This block clear routine may be used to store zeros in a block of registers. BC uses the complete form of the general block control word bbb.eeeII and can thus be used to clear blocks of consecutive registers or can be used to skip over registers within a block.

Example 1: Use BC to clear registers R05-R15.

Before clearing these registers we will first use the block increment routine BI to load them with data. (The PPC ROM routines BI and BV are extremely convenient for loading and viewing blocks of registers). Key 5.015 ENTER↑ 1 ENTER↑ and XEQ "BI". The consecutive integers from 1-11 should now be loaded in R05-R15 inclusively. To convince yourself, clear flags F09 and F10 and key 5.015 XEQ "BV". The block view routine BV should run through the registers and show the contents as described.

Now to clear these registers, simply key 5.015 and XEQ "BC". If you again use BV to view these registers you won't see anything because they have been cleared.

Example 2: Use BC to clear every 5th register starting with R07 and ending with R102.

Assuming you have the available memory and that the current size is at least 103, simply key in 7.10205 and XEQ "BC". The registers R07, R12, R17, R22, R27, R32, R37, R42, R47, R52, R57, R62, R67, R72, R77, R82, R87, R92, R97, and finally R102 should all contain zero. These registers may be inspected by keying in 7.10205 and XEQ "BV". Since BV skips over registers which are zero, none of the above registers will show up in the display; only a series of short TONES will be heard as BV runs through the registers.

COMPLETE INSTRUCTIONS FOR BC

1) The only input to BC is the block control word which is of the form bbb.eeeII.

2) BC contains an internal ISG loop that is controlled by bbb.eeeII. BC stores and uses this block control word in the Last X register. The Y, Z, and T registers are all preserved by BC. The X register contains 0 when BC ends. The following shows the stack contents on input/output from BC.

Input to BC:	Output from BC:
T: T	T: T
Z: Z	Z: Z
Y: Y	Y: Y
X: bbb.eeeII	X: 0
L: L	L: final control word

MORE EXAMPLES OF BC

Example 3: Use BC to clear registers R13-R49 inclusive.

Key 13.049 and XEQ "BC".

Example 4: Use BC to clear the even numbered registers from R02-R100.

Key 2.10002 and XEQ "BC".

Example 5: Use BC to clear the odd numbered registers from R01-R99.

Key 1.09902 and XEQ "BC".

Routine Listing For: BC	
208♦LBL "BC" 209 SIGN 210 CLX 211♦LBL 13 212 STO IND L 213 ISG L 214 GTO 13 215 RTN	

LINE BY LINE ANALYSIS OF BC

BC is a very short routine. The SIGN function at line 209 is used to store the block control word in LAST X which is used as an ISG counter in the loop in BC.

CONTRIBUTORS HISTORY FOR BC

The BC routine and documentation were written by John Kennedy (918) based on the suggestion from Richard Schwartz (2289) that where possible, the block routines should make full use of the block control word.

FINAL REMARKS FOR BC

The one area of improvement for a future BC routine would be greater speed.

FURTHER ASSISTANCE ON BC

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 evenings

NOTES

BC

NOTES

TECHNICAL DETAILS

XROM: 20, 43 **BC** SIZE: depends on block used

Stack Usage:
- 0 T: not used
- 1 Z: not used
- 2 Y: not used
- 3 X: 0
- 4 L: ISG counter

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
R00:

R06: R07: R08: R09: the data registers used are those defined by the block

R10:
R11:
R12:

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine: 13

Execution Time: depends on block size, clears approximately 7.8 registers per second

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: M2

Bytes In RAM: 18

Registers To Copy: 61

Other Comments:
No special SIZE requirement is necessary provided the data block already exists

BD - BASE B TO BASE DECIMAL

This base conversion routine is from base b to base 10 where the base b lies in the range 2<=b<=25. The base b remains stored in a data register and BD takes its input from the alpha register where this routine takes advantage of the alpha capabilities when b>10. See also the routine TB. This routine is the inverse of TB.

Example 1: Convert 43AB (hexadecimal base 16) to base ten.

Store the base 16 in R06. 16 STO 06. Go into alpha mode and key in the characters 43AB. Switch out of alpha and XEQ " BD ". The result base ten, 17323 is left in the X-register.

Example 2: Convert 10110110 (binary, base 2) to base ten.

Store the base 2 in R06. 2 STO 06. Go into alpha mode and key in the characters "10110110". Switch out of alhpa and XEQ " BD ". The result base ten, 182, is left in the X-register.

COMPLETE INSTRUCTIONS FOR BD

1) To convert an integer from base b (where 2<=b<=25) to base ten, first store the base b in R06.

2) Key in the base b digits of the number in the alpha register. Up to 14 digits may be input. Any digit input must be strictly less than the base b.

(Note: the base ten equivalents of the alphabet start with A=10, B=11, C=12, D=13, E=14, F=15, G=16, H=17, I=18, J=19, K=20, L=21, M=22, N=23, O=24. Numeric digits 0-9 are shifted characters in alpha mode. Do not confuse the letter O with the number 0.)

3) Go out of alpha mode and XEQ " BD ". The base ten result will be returned in the X-register. The original number in alpha is not preserved and the alpha register is left cleared. The base b remains stored in R06 for subsequent conversions. The routine performs exact integer arithmetic but it is possible to cause overflow if a combination of a large enough base or a large enough number of digits are input.

The stack input/output for BD is as follows:

Input:	Output:
T: T	T: scratch
Z: Z	Z: 0
Y: Y	Y: 0
X: X	X: base 10 result
L: L	L: 9
M: } digits of the	M: } alpha
N: } number base b	N: } is
O: blanks	O: } cleared

MORE EXAMPLES OF BD

Example 3: Convert FFFF base 16 to base 10.

Store 16 in R06. Key "FFFF" in alpha, go out of alpha and XEQ " BD ". The answer, base ten = 65535.

Example 4: Computer programs running under the standard CP/M operating system start executing at memory location 100H (H is not a digit but represents the hexadecimal base 16). What is this number in base ten?

The base 16 should still remain in R06 from the previous example. Key "100" in alpha and then XEQ " BD ". The answer, base ten = 256.

Example 5: Find the decimal equivalent of seven O's (letter "oh") when b=25.

Store 25 in R06. Key seven O's in alpha and XEQ " BD ". Answer = 6103515624.

Routine Listing For: **BD**

```
01♦LBL "BD"        19 -
02♦LBL A           20 X>0?
03 CLST            21 DSE X
04♦LBL 01          22 9
05 "⊦ "            23 +
06 X<> ]           24 X<0?
07 X=0?            25 GTO 02
08 GTO 01          26 X<>Y
09 X<> [           27 RCL 06
10 R↑              28 *
11 X<> \           29 +
12 "⊦♦å"           30 .
13 X<> \           31 GTO 01
14 RDN             32♦LBL 02
15 X<> [           33 RDN
16  E              34 CLA
17 *               35 RTN
18 39
```

LINE BY LINE ANALYSIS OF BD

Lines 01-02 The purpose of LBL A is to automatically assign BD to key A when the program pointer is stopped in this section of ROM.

Line 03 clears the stack and initializes the main program loop which starts at LBL 01. The base 10 number will be accumulated in the Y register.

Lines 04-31 are the main program loop. Lines 04-08 serve the purpose of appending blanks in alpha until the first character of the number the user has keyed in alpha is pushed into the O register. At the start of LBL 01, X, Z, and T contain 0's and the base 10 number is in Y. Once in the loop, the next character which appears in O is exchanged with M (line 09) and the stack is rolled up (line 10) to preserve M in the stack. The N register is then brought into X and O is stored in N (line 11). The alpha register (MNO) now contains only the next character. Line 12 appends 00H and 08H to this character. N is then returned with its original contents (line 13), the stack is rolled down and then line 15 returns the M register to its original contents and our next character now appears as a decimal number in X. Lines 16 and 17 multiply this number by 1 so it becomes normalized and then 39 is subtracted from this so we may test whether our character is a digit 0-9 (row 3 in the HEX TABLE) or a letter (row 4 in the HEX TABLE). Lines 21-23 then transform the X-register to its true decimal value. Lines 24-25 test if this is a valid number (a blank would yield a negative at this point causing a jump to LBL 02). If not a blank, the accumulated result is multiplied by the base (lines 27 & 28) and the next digit is added (line 29). Line 30 ensures the Y register is the only nonzero stack register when the jump is made back to LBL 01 (line 31).

Lines 32-35 end the routine by rolling down the stack to bring the Y register result into X and the alpha register is cleared when the routine ends.

REFERENCES FOR BD

1. HP-25 Library "65 NOTES" V4N4P8b.
2. George Eldridge (5575) "PPC CALCULATOR JOURNAL" HP-41 HEX TO/FROM DECIMAL V7N9P31b.

CONTRIBUTORS HISTORY FOR BD

George Eldridge (5575) wrote the BD routine. John Kennedy (918) provided the documentation.

FINAL REMARKS FOR BD

A future base conversion routine would profit from greater speed. The implementation of BD on the HP-41C is probably optimal.

FURTHER ASSISTANCE ON BD

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 evenings

NOTES

T E C H N I C A L   D E T A I L S

XROM: 20, 17 | **BD** | SIZE: 007 (minimum)

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: used
- 6 N: used
- 7 O: used
- 8 P: used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
- R00:
- R06: base b
- R07: BD does not use
- R08: any registers other than R06
- R09:
- R10:
- R11:
- R12:

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine: A, 01, 02

Execution Time: 2.2 seconds minimum plus approximately 0.65 seconds per digit (character)

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: BD

Bytes In RAM: 58

Registers To Copy: 53

Other Comments:

BE

BE - BLOCK EXCHANGE

This routine may be used to exchange any two blocks of data registers. The two blocks are described by block control words of the form bbb.eeell. This routine is an extension of the primary-secondary register exchange function that was on the HP-67/97. The two blocks are completely independent and may be of different sizes or they may even overlap.

Example 1: Use BE to exchange R00-R09 with R10-R19.

SIZE 020 minimum. To explicitly show the exchange we will first use BI to load R00-R19 with consecutive integers from 0-19. Key 0.019 ENTER↑ 0 ENTER↑ 1 XEQ "BI". The data loaded in R00-R19 may be verified (first clear flags F09 and F10) by keying 0.019 XEQ "BV". The block view routine BV should show the contents as follows:

R00: 0	R10: 10
R01: 1	R11: 11
R02: 2	R12: 12
R03: 3	R13: 13
R04: 4	R14: 14
R05: 5	R15: 15
R06: 6	R16: 16
R07: 7	R17: 17
R08: 8	R18: 18
R09: 9	R19: 19

Now to exchange these two blocks, simply key 0.009 ENTER↑ 10.019 XEQ "BE". As can be verified by keying 0.019 XEQ "BV", the following shows these registers with their contents exchanged.

R00: 10	R10: 0
R01: 11	R11: 1
R02: 12	R12: 2
R03: 13	R13: 3
R04: 14	R14: 4
R05: 15	R15: 5
R06: 16	R16: 6
R07: 17	R17: 7
R08: 18	R18: 8
R09: 19	R19: 9

COMPLETE INSTRUCTIONS FOR BE

1) The only input to BE is the two block control words which describe the blocks to be exchanged. Both block control words are assumed to be of the standard form bbb.eeell which the 41C uses for its ISG and DSE functions. BE contains an internal ISG loop. The block control words will normally describe independent blocks, but the blocks can overlap.

2) Key in the two block control words in the Y and X registers. For most applications the two blocks will be of the same size in which case the order of the block control words input is not important. However, when the blocks are of different sizes, the ISG function that determines when BE ends is controlled by the block control word that is entered in the Y-register. The block control word in the X-register is used to increment the X-register block count but this control word is not used to test the end of the block.

3) XEQ "BE" and the two blocks will be exchanged by starting with the first register of each block. The routine preserves LAST X and Z but T is lost and X and Y contain the final block control words when BE ends.

MORE EXAMPLES OF BE

Example 2: Use BE to exchange the even numbered registers in the two blocks in Example 1.

First key 0.019 ENTER↑ 0 ENTER↑ 1 XEQ "BI" to ensure R00-R19 are in their original order.

R00: 0	R10: 10
R01: 1	R11: 11
R02: 2	R12: 12
R03: 3	R13: 13
R04: 4	R14: 14
R05: 5	R15: 15
R06: 6	R16: 16
R07: 7	R17: 17
R08: 8	R18: 18
R09: 9	R19: 19

These two blocks are the same length and the exchange is made on every other register. The two block control words are 0.00902 and 10.01902. Key in these two block control words in Y and X and then XEQ "BE". The exchange that takes place is reflected in the following list of the above registers. Verification can be made by using the BV routine to view these registers. Key 0.019 XEQ "BV".

R00: 10	R10: 0
R01: 1	R11: 11
R02: 12	R12: 2
R03: 3	R13: 13
R04: 14	R14: 4
R05: 5	R15: 15
R06: 16	R16: 6
R07: 7	R17: 17
R08: 18	R18: 8
R09: 9	R19: 19

Example 3: This example will show the use of BE to exchange parts of blocks which overlap. Exchange registers R20-R28 with registers R25-R33.

R20: 12	R25: 33	Note that R25-R28 are listed twice.
R21: 71	R26: 19	
R22: 13	R27: 24	
R23: 30	R28: 93	
R24: 14	R29: 75	
R25: 33	R30: 55	
R26: 19	R31: 67	
R27: 24	R32: 97	
R28: 93	R33: 85	

To perform the block exchange key 20.028 ENTER↑ 25.033 and XEQ "BE". The following shows the contents of the blocks after the exchange has been made.

R20: 33	R25: 55	Note that R25-R28 are listed twice.
R21: 19	R26: 67	
R22: 24	R27: 97	
R23: 93	R28: 85	
R24: 75	R29: 14	
R25: 55	R30: 12	
R26: 67	R31: 71	
R27: 97	R32: 13	
R28: 85	R33: 30	

Routine Listing For:	BE

```
32+LBL "BE"
33 RCL IND Y
34 X<> IND Y
35 STO IND Z
36 RDN
37 ISG X
38 ""
39 ISG Y
40 GTO "BE"
41 RTN
```

LINE BY LINE ANALYSIS OF BE

BE is a very short routine. At line 32 the two block control words are assumed to be in the stack in X and Y. Lines 33-35 perform the exchange on an element by element basis as part of the loop in the program. The RDN instruction at line 36 puts the stack back in the correct configuration for the next pass through the loop. Both X and Y are incremented but note that only Y is tested. Line 38 is a NOP.

REFERENCES FOR BE

John Kennedy "PPC Calculator Journal" ROM Progress V7N3P5

CONTRIBUTORS HISTORY FOR BE

The BE routine is by Richard Schwartz (2289) who suggested using a full block control word for each of the two blocks. John Kennedy (918) provided the documentation.

FINAL REMARKS FOR BE

BE could profit from greater speed, but this is more of a shortcoming of the calculator in general rather than the implementation of BE.

FURTHER ASSISTANCE ON BE

John Kennedy (918) phone: (213) 472-3110 evenings

Richard Schwartz (2289) phone: (213) 447-6574 eve.

NOTES

TECHNICAL DETAILS

XROM: 20, 34 — BE — SIZE: depends on blocks used

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊦: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:

R00:

R06: The data registers used depend on the two blocks of data

R07:

R08:

R09:

R10:

R11:

R12:

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine: none

Execution Time: depends on block size, exchanges approximately 0.9 registers per second

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: M2

Bytes In RAM: 25

Registers To Copy: 61

Other Comments:

No special SIZE requirement is necessary provided the data blocks already exist

BI - BLOCK INCREMENT

BI is a register block operation that may be used to load a block of registers with numerical data. If the data to be loaded is zero BI may be used to clear registers. If the increment input value is zero BI may be used to load a constant into a block of registers.

BI

Example 1: Store the numbers 1 - 10 into registers 21 - 30. (SIZE must be ≥31.) FIX 3.

DO:	SEE:	RESULT:
XEQ "CLRG"	No change	Data registers cleared.
21.03, ENTER	21.030	bbb.eeeii- input for BI.
1, ENTER	1.000	Start value input for BI.
1	1.000	Increment value input for BI.
XEQ BI	10.000	BI finished, last value stored is in the display.

Example 2: Store -10, -15, -20, -25, -30 into registers 13 thru 17. FIX 3.

DO:	SEE:	RESULT:
XEQ "CLRG"	No change	Data registers cleared.
13.017, ENTER	13.017	bbb.eeeii input for BI.
-10, ENTER	-10.000	Start value input for BI.
-5	-5.000	Increment value input BI.
XEQ BI	-30.000	BI finished, last value stored is in the display.

Example 3: Store 5000, 4500, 4000, 3500, 3000, 2500, 2000, 1500, 1000, 500 into register 0 thru 9. FIX 3.

DO:	SEE:	RESULT
.009, ENTER	0.009	(input bbb.eeeii value)
5000, ENTER	5000.000	(input start value)
-500, XEQ BI	500.000	(input increment value)

To verify key: .009, XEQ BV.

COMPLETE INSTRUCTIONS FOR BI

Block Increment is a powerful routine to pre-load registers with data for test, demonstration, or problem solving purposes. Three inputs are required as shown by the stack condition for storing 10 thru 1 in R1-R10. FIX 3.

T:	TTT		T:	TTT	(input T reg.)
Z:	1.010		Z:	TTT	(input T reg.)
Y:	10.000	XEQ BI	Y:	11.010	(Final ISG.)
X:	-1.000	⟶	X:	1.000	(last stored value)
L:	0.000		L:	-1.000	(X input)

The T register is duplicated into Z and last X contains the X input value. X contains the last stored value and Y contains the value (eee +1).(eee).

If the increment value is zero BI, becomes a "constant load" routine that stores the start value into all registers in the block.

If the increment value and the start value is zero, BI becomes a block clear. This use of BI, however, is not the optimum way to clear registers unless the stack preservation characteristics of BI are needed. Here is the relative timing of BI and BC.

REGISTERS CLEARED	10	50	100	150	200	250
USING BI	1.95	8.09	15.70	23.61	31.52	39.38
USING BC	1.49	6.29	12.12	18.17	24.32	30.42

This data shows BI clears 6.2 registers per second compared to BC's 8.1 registers per second. Execution times for the "constant load" mode are the same as the "clear" mode.

Data may be loaded into a block of registers in ascending values (increment value positive) or in descending values (increment value negative). Figure 1. illustrates the various possibilities.

bbb.eee = 0.007

ii	Start	INC.	R00	R01	R02	R03	R04	R05	R06	R07
00	13	1	13	14	15	16	17	18	19	20
00	20	-1	20	19	18	17	16	15	14	13
00	2	2	2	4	6	8	10	12	14	16
00	24	-3	24	21	18	13	12	9	6	3
00	-55	-5	-55	-60	-65	-70	-75	-80	-85	-90

ii	Start	INC.	R00	R01	R02	R03	R04	R05	R06	R07
00	-55	5	-55	-50	-45	-40	-35	-30	-25	-20
02	13	1	13	-	14	-	15	-	16	-
03	20	-1	20	-	-	19	-	-	18	-
07	24	-3	24	-	-	-	-	-	-	21
02	-5	3	-5	-	-2	-	1	-	4	-

FIGURE 1. Examples of BI capabilities.

If BI is used in a program the user may be tempted to omit the ENTER instructions as shown on the left to save bytes. There is no difference in the byte count because the 41 "operating system" places NULLS in place of the ENTERS anyway. The NULLS may look simpler, but the ENTERS will be less confusing to most users—especially those unfamiliar with entry termination.

```
LBL A         LBL B
1.01          1.01
 10           ENTER
 -1            10
XROM BI       ENTER
RTN            -1
              XROM BI
              RTN
18 Bytes OK   18 Bytes BETTER
```

If the start value and the increment value are both the same the value need only be entered once, e.g., store 1-10 in R30-R39 would be:

30.039, ENTER, 1, ENTER, XROM BI

MORE EXAMPLES OF BI

Example 4: Janet wants to test a sort program and decides to use the PPC ROM Block Increment routine to load 100 test data. She decides that the data should meet the following requirements.

a. 50% of data in ascending order.
b. 50% of data in descending order.
c. Data to be of highest three digit values.
d. Ascending data should be odd.
e. Descending data should be even.
f. Data should be intermixed.
g. Data should be in R03-R102.

The largest 100 three digit data means numbers running from 900 to 999. This dictates that R102 = 999, R101 = 900, R03 = 998 and R04 = 901. Janet's program to load the specified data is:

```
01 LBLTLOAD        08 3.10202
02 4.10202         09 ENTER
03 ENTER           10 998
04 901             11 ENTER
05 ENTER           12   -2
06   2             13 XROM BI
07 XROM BI         14 RTN
```

She decides to use BV and a printer to make a quick check of the data. This program is shown below. After testing the sort program (it wasn't as fast as S3) she decides to compare data that doesn't meet "f" of intermixed. She reloads the data by changing line 02 from 4.10202 to 53.102 and line 08 from 3.10202 to 3.052. This places the ascending data in one half of the 100 register block and the descending data in the other half.

```
01♦LBL "LOAD"    12 -2          3: 998    45: 914    90: 975
02 53.102        13 XROM "BI"   4: 996    46: 912    91: 977
03 ENTER↑        14 BEEP        5: 994    47: 910    92: 979
04 901           15 3.01        6: 992    48: 908    93: 981
05 ENTER↑        16 XROM "BV"   7: 990    49: 906    94: 983
06 2             17 ADV         8: 988    50: 904    95: 985
07 XROM "BI"     18 90.102      9: 986    51: 902    96: 987
08 3.052         19 XROM "BV"   10: 984   52: 900    97: 989
09 ENTER↑        20 XROM "PO"             53: 901    98: 991
10 998           21 RTN                   54: 903    99: 993
11 ENTER↑        22 .END.                 55: 905    100: 995
                                                     101: 997
                                                     102: 999
```

FURTHER DISCUSSION OF BI

BI requires three inputs and has the full power of ISG with the ii portion of the block definition of bbb.eeeii. The input order is easily remembered if your thinking goes along these lines.

I need data in registers. Which registers?

1st Input is bbb.eeeii

BI increments the data. What value does the data start with?

2nd Input is start value.

The data is incremented from the start value. What is the increment value?

3rd Input is increment value.

NOW XEQ BI

Routine Listing For: BI

```
61♦LBL "BI"      66 STO IND Y
62 -             67 ISG Y
63♦LBL 10        68 GTO 10
64 LASTX         69 RTN
65 +
```

CONTRIBUTORS HISTORY FOR BI

BI was written by Roger Hill (4940) specifically for the PPC ROM

FINAL REMARKS FOR BI

BI is a classic example of a short efficient routine that greatly expands the capability of the basic machine. BI , like all of the block operations, should be included in all future PPC's. If BI is in 'microcode' it could run faster. Improvements might be made if the block control number is returned to x upon completion. Greater flexibility could be obtained by recognizing an "ALPHA" input form that would perform an alpha constant load. Input in this case would be bbb.eee, ENTER, alpha string, XEQ BI .

FURTHER ASSISTANCE ON BI

Richard Nelson (1) (714) 754-6226 P.M.

T E C H N I C A L   D E T A I L S

XROM: 10,44	BI	SIZE: AS REQUIRED

Stack Usage: 0 T: NOT USED 1 Z: USED 2 Y: USED 3 X: USED 4 L: USED	Flag Usage: NONE 04: 05: 06: 07: 08:
Alpha Register Usage: 5 M: NOT USED 6 N: NOT USED 7 O: NOT USED 8 P: NOT USED	09: 10: 25:
Other Status Registers: 9 Q: 10 ⊢: 11 a: 12 b: 13 c: 14 d: 15 e: NONE USED	Display Mode: N/A Angular Mode: N/A Unused Subroutine Levels: 5
ΣREG: NOT USED Data Registers: R00: R06: R07: R11: R12: As required by data.	Global Labels Called: Direct / Secondary NONE Local Labels In This Routine: LBL 10
Execution Time: 4.7 to 6.1 Registers per second.	
Peripherals Required: NONE	
Interruptible? YES Execute Anytime? NO Program File: BL Bytes In RAM: 19 Registers To Copy: 46	Other Comments:

BL - BLDSPEC INPUTS FOR LB

BL makes it easy to use completely arbitrary graphics characters in print formats, storing the character codes very compactly in the program itself. This contrasts with the conventional BLDSPEC method which can only store the character on a data card, or use a very cumbersome series of program lines (up to 35 bytes). The BL routine output is the character code expressed as a sequence of decimal byte numbers, ready for use in LB, which creates a synthetic text program line defining the desired character.

Example 1: Print a line beginning with the special graphics character having the dot pattern shown below.

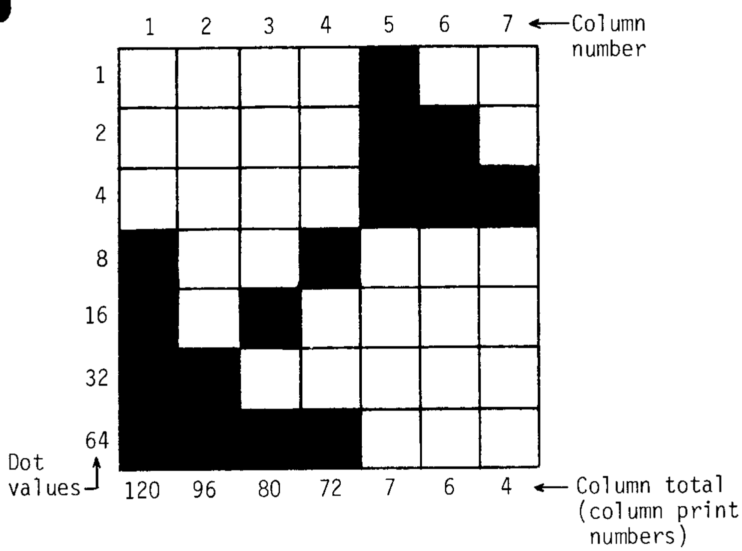


Figure 1.

1) Determine the BLDSPEC column totals shown, by adding the value of the desired dots for each of the seven columns.

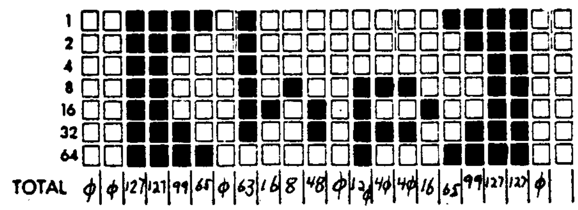
2) Use BL to convert them into seven decimal byte numbers, (the "character code") for input to LB:

DO:	SEE:	RESULT (RECORD)
120 XEQ BL	17	First decimal byte number
96 R/S	227	Second " "
80 R/S	5	Third " "
72 R/S	9	Fourth " "
7 R/S	1	Fifth " "
6 R/S	195	Sixth " "
4 R/S	4	Seventh " "

3) Use LB to generate a 7-character text line, using the prefix byte F7 (decimal byte 247) followed by the above seven decimal bytes: (247, 17, 227, 5, 9, 1, 195, 4). The resulting program line displays (result of executing shown in parentheses):
01 ⊺ ⊠ ⊠ ⊼ ⊠ ⊼ ⊠ ⊼ (character code into M register)

4) Write the rest of the program:
02 RCL M (character code into X register)
03 ACSPEC (loads character into print buffer)
04 PRBUF (prints contents of print buffer)
When run, this program causes printing of the desired special graphics line. (see XL for loading lines 03 and 04 when the printer is not present.)

Example 2: A fast, non-register usage, routine is desired to print the HP LOGO. The desired printer "character" is sketched below.



BL is used to convert the BLDSPEC "numbers" into the bytes to be placed into program memory. (Using LB) in the format:

```
01  LBL T HP LOGO
02  First BLDSPEC "character" text line
03  RCL M
04  ACSPEC
05  Second BLDSPEC "character" text line
06  RCL M
07  ACSPEC
08  Third BLDSPEC "character" text line
09  RCL M
10  ACSPEC
11  PRBUF
12  STOP
```

Lines 02, 05, and 08 are determined using BL as shown.

DO	SEE	RESULT
0	XEQ BL	16, first LB text byte for line 02
0	R/S	0, second LB text byte
127	R/S	7, third
127	R/S	255, fourth
99	R/S	248, fifth
65	R/S	224, sixth
0	R/S	128, seventh
63	XEQ BL	16, first LB text byte for line 05
16	R/S	252, second
8	R/S	128, third
48	R/S	134, fourth
0	R/S	0, fifth
120	R/S	60, sixth
40	R/S	40, seventh
40	XEQ BL	16, first LB text byte for line 08
16	R/S	160, second
65	R/S	132, third
99	R/S	28, fourth
127	R/S	127, fifth
127	R/S	255, sixth
0	R/S	128, seventh

The three text lines must be preceeded by a text 7, byte 247. Using the guidelines described in the LB section and the HP-41C COMBINED HEX/DECIMAL BYTE TABLE the LB inputs would be:

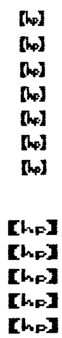
	PROGRAM LINE	DECIMAL LB INPUTS
01	LBL T HP LOGO	192, 0, 248, 0, 72, 80, 32, 76, 79, 71, 79
02	⊠ ⁻ ⊠ ⊠ ⊠ ⊠ ⊠	247, 16, 0, 7, 255, 248, 224, 128
03	RCL M	144, 117
04	ACSPEC (XROM 29, 04)	167, 68*

05	⊠⊠⊠⊠‾<(	247, 16, 252, 128, 134, 0, 60, 40
06	RCL M	144, 117
07	ACSPEC (XROM 29,04)	167, 68*
08	⊠⊠⊠⊠⊢⊠⊠	247, 16, 160, 132, 28, 127, 255, 128,
09	RCL M	144, 117
10	ACSPEC (XROM 29,04)	167, 68*
11	PRBUF (XROM 29,10)	167, 74*
12	STOP	132
		50 Bytes

*XROM instructions are obtained by using XL. e.g. 29 ENTER 4 XEQ XL gives 167 in X and 68 in Y. (Use X≷Y to see second byte).

The routine and it's printed output is shown below. Execution speed is less than 1 second.

```
 01♦LBL "HP
LOGO"
 02 "θ♦↓
"
 03 RCL [
 04 ACSPEC
 05 "θ♦<("
 06 RCL [
 07 ACSPEC
 08 "θ∝⊢"
 09 RCL [
 10 ACSPEC
 11 PRBUF
 12 GTO "HP
LOGO"
 13 END
```



COMPLETE INSTRUCTIONS FOR BL

BL converts a sequence of seven graphics character column totals to the seven decimal byte numbers which LB needs in order to include the character code as a text line in a program. Display mode FIX 0 is convenient.

1) Lay out the desired dots in the 7 x 7 format, and using the dot values shown in the diagram above, compute the seven column totals.

2) Key in the first column total, XEQ BL, and record the result, DB1 (decimal byte 1).

3) Key in the second column total, R/S, record the result, DB2. Repeat for third to seventh numbers.

4) Use LB and the eight byte numbers (247, DB1, DB2, DB3, DB4, DB5, DB6, DB7) to create a seven-byte text line.

In a program, executing that text line places the character code into the M register. The M register contents are now in "alpha data" format (first nybble is 1) so its contents can be transferred intact by functions RCL M, STO nn, X<>nn, etc. as desired. To use the character in a print format, place its code in the X register by a RCL, X<> or stack transfer. Then ACSPEC appends the character to the contents of the print buffer for eventual printing by ADV or PRUBUF.

BL calls subroutine QR, uses the stack plus M and O registers, and uses no data registers or flags.

LINE BY LINE ANALYSIS OF BL

The character code placed in the M register by line 01 of Example 1 is shown below as a 56-bit binary string. Examination of the column print numbers above the string shows that the bits correspond to the dots, arranged by columns.

```
                 column print numbers                7 BITS
          | 120  |  96  |  80  |  72  |   7  |   6  |   4
00010001111000110000010100001001000000011100001100000100
   17    | 227  |   5  |   9  |   1  | 195  |   4
                 decimal byte numbers                8 BITS
```

The first nybble (4 bits) is the alpha data prefix, insuring that the character code can be transferred between registers without change. The output of BL is the sequence of decimal bytes shown below the bit string, obtained by partitioning each column print number and combining the upper part with the part left over from the preceding partition.

Lines 01 - 06 start with the first column print number (120) in the X register, and set up the M, X, and Y registers for the first of seven passes through LBL 02.

At Line 07 on each pass the X register contains the entered column print number, the Y register contains the right hand part of the preceding column print number, and M contains a power of 2 which controls the partitioning. On the first pass M = 2 and Y = 16 (acting like a preceding right-hand part) for setting up the alpha data prefix (leading 0001 nybble).

Lines 8 - 11 construct the partitioning divisor in X (lifting the stack) and double the number in M.

Line 12 does the partitioning: Y/X (120/64) puts the remainder (56) in X and the integer quotient (1) in Y. QR uses and clears the O register, leaves the divisor in L.

Lines 13 to 17 add the quotient to the previous right-hand part and stop, showing the result (17) in X. They also multiply the current remainder (by 4 in this pass for a two-place binary shift) preparing it for the next pass addition.

After recording the result, the next column print number must be entered, and R/S restarts.

Lines 18 - 20 put the binary-shifted remainder (224) into Y, ready for the next pass via LBL 02.

Routine Listing For: BL	
01♦LBL "BL"	11 /
02 2	12 XROM "QR"
03 STO [	13 RCL [
04 X↑2	14 *
05 X↑2	15 X<> Z
06 X<>Y	16 +
07♦LBL 02	17 STOP
08 128	18 X<>Y
09 RCL [	19 RDN
10 ST+ [	20 GTO 02

REFERENCES FOR BL

PPC CALCULATOR JOURNAL, V7N5P56, is the basic article on the synthetic programing method, replacing BLDSPEC.

PPC CALCULATOR JOURNAL, V7N6P24 and V7N6P27 cover wand techniques for circumventing BLDSPEC inconveniences, but they have been superseded by the BL - LB team. (Interestingly, contrary to V7N6P24a, the old original BLDSPEC could actually create any synthetic character alpha data in the X register.)

PPC CALCULATOR JOURNAL, V8N2P41, has a compact description of BL.

CONTRIBUTORS HISTORY FOR BL

Keith Jarett (4360) wrote BL, automating the manual procedure described by William C. Wickes (3735) in the first reference above.

FURTHER ASSISTANCE ON BL

Call Keith Jarett (4360) at (213) 374-2583.
Call Richard Nelson (1) at (714) 754-6226 P.M.

NOTES

TECHNICAL DETAILS

XROM: 10,42 | **BL** | SIZE: 000

Stack Usage:
- 0 T: USED
- 1 Z: USED
- 2 Y: USED
- 3 X: USED
- 4 L: USED

Alpha Register Usage:
- 5 M: USED
- 6 N: USED
- 7 O: USED
- 8 P:

Other Status Registers:
- 9 Q:
- 10 ⊢:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

NONE USED BY ROUTINE

ΣREG: NOT USED

Data Registers:

R00:

R06:
R07:
R08:
R09:
R10:
R11:
R12:

NONE USED BY ROUTINE

Flag Usage: NONE

04:
05:
06:
07:
08:
09:
10:

25:

Display Mode: N/A

Angular Mode: N/A

Unused Subroutine Levels: 4*

Global Labels Called:

Direct	Secondary
QR	NONE

Local Labels In This Routine:

LBL 02

Execution Time: < 1 Second per input.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	BL
Bytes In RAM:	37
Registers To Copy:	46

Other Comments:

* This routine is not intended to be called as a subroutine.

APPENDIX A CONTINUED FROM PAGE 37.

07 partition data
08 also partition data, if # of parts > 5

The global work area is accessible to GHT regardless of the depth of recursive call ($\le n_3-1$). The specification $W.pm_0$ is a compact storage of three items of information needed by MOVE (the subroutine for moving one disc to peg Y from peg X) and SHOW (the subroutine for displaying the current distribution of discs on pegs):

W = pointer to global work area

p = # of data registers allocated to each peg

m_0 = original number of pegs ($\underline{m}$ passed to IGT)

Partition data is in a compact form ($a_1a_2b_1b_2\ldots$), a pair of decimal digits to each part, beginning with number of discs to be moved using 3 pegs, and ending with the number of pegs to be moved using m'-2 pegs. (This data is set up in lines 03 through 22 of GHT; register 08 is only needed when m'-2 > 5 or m' > 7, but to avoid the logic overhead GHT always uses 9 registers per recursive call.) The position of the decimal point varies during the process. When loading the intermediate pegs, the decimal point moves to the left; it moves back to the right when the intermediate pegs are being unloaded.

The global work area is allocated as follows:

W: move counter, initialized to -1

W+1 → W+p: discs for peg 1

W+p+1 → W+2p: discs for peg 2

⋮

$W+(m_0-1)p+1 \rightarrow W+m_0p$: discs for peg m_0

Discs are designated by integers from 1 to n, where i<j whenever disc i is smaller than disc j. Each disc designation i is kept in compact storage (at most 5 to a register) as two decimal digits $d_{i1}\ d_{i2}$.

IGT initializes the work area and R01 through R04, given the number ($\underline{m}$) of pegs in register Y and the number ($\underline{n}$) of discs in register X:

.12---m ---> R01

n ---> R02

m ---> R03

W.pm ---> R04

-1 ---> R_W

$d_{11}d_{12}d_{21}d_{22}$---$_{51}d_{52}$ ---> R_{W+1}

⋮

---$d_{n1}d_{n2}$ --- R_{W+p}

(in fact, IGT begins with a CLRG, so any register not explicitly addressed in IGT starts out with a zero value.) Additionally IGT calls on IXR to initialize return stack management. (See Application Program 1 in LR description for further details regarding IXR.) Note that the pointer in R13 is set to 9 less than pointer in R04 before GHT calls itself. (See lines 231 through 234.) Of course, before a call on itself GHT must also set up R10 through R12 which become R01 through R03 after the curtain is raised. (Lines 51 through 83 do this during the loading of the intermediate pegs; lines 139 through 193 do the same task for the unloading process.)

Finally, to avoid loss of a return path as a consequence of excessive subroutine nesting, GHT calls LRR upon entry and SRR just before exit. No other calls for safeguarding the return path are necessary, since GHT does not initiate any other chain of calls more than two deep. (See Application Program 1 in LR description for further details regarding LRR and SRR.) However, a brief examination of the Q(m,e) table will show that the two cases with the smallest number of moves that require an extension of the return stack ($n_3 \ge 6$) are $\underline{m}$ = 3 and $\underline{n}$ = 7 or 8, which entail 127 and 255 moves respectively. Other stack-extending cases are far more prolonged. If you plan to avoid such time consuming cases (by restricting yourself to cases where $n_3 < 6$) you can avoid the execution overhead of IXR, LRR, and SRR by removing lines 80 through 81 in IGT, lines 02 and 218 through 219 in GHT, and replace 'GTO 15' in line 165 of GHT by "RTN".

The logic of 'MOVE' and 'SHOW' (disc stacks are displayed from top to bottom), although using some tedious housekeeping to unravel compact storage, can be gleaned by careful perusal of the listing, keeping in mind the allocation scheme already described. However, a few words about 'PARTS' are needed to ease comprehension of its logic.

If we examine the Q(m,e) table, a simple method for evaluating the optimum distribution of discs on intermediate pegs quickly becomes apparent. We'll use an earlier example of m=6 and n=25 to keep our description concrete. 'PARTS' builds up the partitioning using R09 through R(6+m), which would be R09 through R12 in our example. We begin with all parts set to zero, and the count $\underline{k}$ of discs to distribute to n-1=24.

```
REPEAT WHILE k > 0:
    INC ← 1
    REPEAT FOR j = 9 through 12:
        Rj ← Rj + INC
        k  ← k  - INC
        INC ← Rj
        IF k=0, EXIT
        IF k<0, Rj← Rj + k and EXIT
```

The following table shows the changing states of R09 through R12 and of $\underline{k}$:

R09	R10	R11	R12	k
0	0	0	0	24
1	0	0	0	23
1	1	0	0	22
1	1	1	0	21
1	1	1	1	20
2	1	1	1	19
2	3	1	1	17
2	3	4	1	14
2	3	4	5	10
3	3	4	5	9
3	6	4	5	6
3	6	10	5	0

A final word of caution. You may want to abort program execution for some reason. If you note your size (via S?, e.g.) before execution, then XEQ S? if you stop the program before it finishes execution, subtract the original size, and call CU to re-establish communication with all your data registers.

APPENDIX A CONTINUED ON PAGE 133.

BM - BLOCK MOVE

The block move routine applies to any block of consecutive data registers. The routine will move the block either forwards or backwards anywhere within the defined data register area. Input to BM requires the number of the first register in the block, the register number which will be the destination of the first register, and the number of registers in the block. The block moved may overlap on itself.

Example 1: Use BM to move the contents of registers R10-R20 to the registers R35-R45.

In this example we will assume the following data are in R10-R20. The data in R35-R45 will be lost, but the data moved from R10-R20 will still remain in R10-R20. Store the following data in R10-R20.

R10: 23
R11: 47
R12: 13
R13: 16
R14: 17
R15: 27
R16: 34
R17: 55
R18: 62
R19: 78
R20: 36

Key in the following to perform this block move. 10 ENTER↑ 35 ENTER↑ 11 and XEQ "BM". The stack input is in the following form:

T: *
Z: 10 = 1st register in block to be moved
Y: 35 = destination of 1st register
X: 11 = number of registers in the block

After BM finishes the following will be the contents in R10-R20 as well as R35-R45.

R10: 23	R35: 23
R11: 47	R36: 47
R12: 13	R37: 13
R13: 16	R38: 16
R14: 17	R39: 17
R15: 27	R40: 27
R16: 34	R41: 34
R17: 55	R42: 55
R18: 62	R43: 62
R19: 78	R44: 78
R20: 36	R45: 36

These two blocks may be inspected by employing the block view routine BV. Key 10.02 XEQ "BV" and key 35.045 XEQ "BV".

COMPLETE INSTRUCTIONS FOR BM

1) The only input to BM is indicated in the following stack registers:

T: *
Z: 1st register in block to be moved
Y: destination register of 1st register
X: number of registers to be moved

2) The block is moved by starting with either the first or last register in the block, depending on whether the first register is moved to a lower or higher numbered register. If the original block and the destination block do not overlap then the original block will be preserved and the routine BM is then equivalent to a block copy in another part of memory. The XYZT stack registers as well as Last X are used by the BM routine. If it is desired to preserve the stack then calls to the ROM routines SM and MS should be made before and after BM is called.

MORE EXAMPLES OF BM

Example 2: Move the contents of R30-R42 to R00-R12.

Key 30 ENTER↑ 0 ENTER↑ 13 and XEQ "BM".

Example 3: Move the block R15-R32 to the registers occupied by R25-R42.

Key 15 ENTER↑ 25 ENTER↑ 18 and XEQ "BM".

Routine Listing For: BM

```
103♦LBL "BM"        115♦LBL 04
104 SIGN            116 R↑
105 RDN             117♦LBL 05
106 X<Y?            118 RCL IND Z
107 GTO 04          119 STO IND Z
108 LASTX           120 RDN
109 ST+ Z           121 ST+ Z
110 +               122 ST+ Y
111 -1              123 DSE L
112 ST+ Z           124 GTO 05
113 ST+ Y           125 RTN
114 RDN
```

LINE BY LINE ANALYSIS OF BM

The first lines in the BM routine determine the required direction of the move and then set up the appropriate parameters.

Lines 117-124 are the main loop in the program. The stack contents at LBL 05 are:

X: ±1 Y: destination pointer Z: source pointer

The sign of X determines whether the pointers are increased or decreased on each pass through the loop.

The LAST X register is used as the DSE counter that controls the loop.

REFERENCES FOR BM

John Kennedy PPC Calculator Journal "ROM PROGRESS" V7N3P5

CONTRIBUTORS HISTORY FOR BM

The BM routine and documentation were written by John Kennedy (918).

FINAL REMARKS FOR BM

The ROM routines BM and BR are the only two block routines which do not use the full extent of the general block control word bbb.eeeii. BM and BR only use bbb.eee. Routines written to make BM and BR use the ii portion were programmed but were found to be too long to include in the ROM for the added benefit and capability that this feature provides.

FURTHER ASSISTANCE ON BM

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve

NOTES

TECHNICAL DETAILS

XROM: 20, 39	BM	SIZE: depends on block used

Stack Usage:	Flag Usage:
0 T: used	04: not used
1 Z: used	05: not used
2 Y: used	06: not used
3 X: used	07: not used
4 L: used	08: not used
Alpha Register Usage:	09: not used
5 M: not used	10: not used
6 N: not used	
7 O: not used	
8 P: not used	25: not used
Other Status Registers:	**Display Mode:**
9 Q: not used	not used
10 ⊦: not used	
11 a: not used	**Angular Mode:**
12 b: not used	not used
13 c: not used	
14 d: not used	**Unused Subroutine Levels:**
15 e: not used	5
ΣREG: not used	**Global Labels Called:**
Data Registers:	Direct: none / Secondary: none
R00:	
The registers used depend on the BM input parameters	
R06:	
R07:	
R08:	
R09:	
R10:	
R11:	**Local Labels In This Routine:**
R12:	04, 05

Execution Time: depends on block size, moves approximately 4.0 registers per second

Peripherals Required: none

Interruptible? yes	**Other Comments:**
Execute Anytime? no	No special SIZE requirement is necessary provided the data block already exists
Program File: M2	
Bytes In RAM: 41	
Registers To Copy: 61	

BR - BLOCK ROTATE

The block rotate routine applies to any set of consecutive data registers. This routine was inspired by the roll up and roll down functions which apply to the XYZT stack registers. Input to BR is the number of the first register in the block and ±n where n is the number of registers within the block. The sign of n determines the direction of the rotation.

Example 1: Use BR to rotate the contents R10-R20 as indicated below.

R10: 50	R16: 56
R11: 51	R17: 57
R12: 52	R18: 58
R13: 53	R19: 59
R14: 54	R20: 60
R15: 55	

Store the above data in the indicated registers. The block increment routine BI may be used to initially store the data. Key 10.02 ENTER↑ 50 ENTER↑ 1 and XEQ "BI". The first register in this block is R10 and the number of registers is 11. Key 10 ENTER↑ 11 and XEQ "BR". The new contents of R10-R20 are indicated by:

R10: 60	R16: 55
R11: 50	R17: 56
R12: 51	R18: 57
R13: 52	R19: 58
R14: 53	R20: 59
R15: 54	

The final contents may be verified by employing the block view routine BV. Key 10.02 XEQ "BV". The contents of the registers have been moved "down". But this terminology can be ambiguous or confusing depending on how one looks at consecutive registers and how one views the HP-41C memory map. A positive number in X causes the contents of the block to essentially shift into higher numbered registers whereas a negative number in X will cause the contents of the block to shift into lower numbered registers.

Example 2: Assuming the data remains as at the end of Example 1 rotate the contents of R10-R20 a second time.

Key 10 ENTER↑ 11 and XEQ "BR". The new contents of R10-R20 are indicated by:

R10: 59	R16: 54
R11: 60	R17: 55
R12: 50	R18: 56
R13: 51	R19: 57
R14: 52	R20: 58
R15: 53	

COMPLETE INSTRUCTIONS FOR BR

1) BR applies only to a consecutive block of data registers. Input to BR requires the number of the first register in the block and ±n where n is the number of registers in the block.

2) The sign of n determines the direction of the rotation. If n is positive the contents of the block will shift to higher numbered registers and if the sign of n is negative the contents of the block will shift to lowered numbered registers.

3) XEQ "BR". The amount of rotation is by one register each time BR is called.

4) The stack input/ouput for BR is indicated below.

Input to BR:	Output from BR:
T: T	T: *
Z: Z	Z: *
Y: 1st register in block	Y: *
X: ± # registers = n	X: -(sign of n)
L: L	L: *

MORE EXAMPLES OF BR

Example 3: Store the following alpha constants in R15-R18.

R15: X R16: Y R17: Z R18: T

Use BR to perform the equivalent of R↓ and R↑.

To perform R↓ the contents of the majority of registers will shift into lower numbered registers so the sign of n should be negative. Key 15 ENTER↑ 4 CHS and XEQ "BR". The contents of R15-R18 should now be as follows:

R15: Y R16: Z R17: T R18: X

To perform R↑ the contents of the majority of registers will shift into higher numbered registers so the sign of n should be positive. Make sure the contents are as originally listed above and key in 15 ENTER↑ 4 and XEQ "BR". The contents of R15-R18 should now be:

R15: T R16: X R17: Y R18: Z

Example 4: Store the following in R09 and R10.

R09: 9 R10: 10

Key 9 ENTER↑ 2 and XEQ "BR" or key 9 ENTER↑ 2 CHS and XEQ "BR". The contents of R09 and R10 should be reversed in either case. This example shows that when the block consists of only two registers BR is equivalent to a register exchange and is independent of the sign of n.

APPLICATION PROGRAM 1 FOR BR

Routine MOVAV simplifies maintaining moving averages, especially multiple moving averages. BR is used to rotate the data block from which the average or averages are to be drawn. By rotating after each entry, 1) no counters are needed as pointers, and 2) data reviewing and error checks are easier with data stored first to last from block endpoint to block endpoint (not starting and ending the series somewhere in the middle).

In the routine below, 5-, 10-, and 20-day moving averages are maintained on a 20 register data block, with the last entry at the bottom of the block. The data block can of course be longer than the averages: just change line 04 to STO nn, RCL 32, where nn is the highest register in the block, and change line 07 from 20 to the size of your block. Negative block size entry will allow you to keep data in the opposite order, with corresponding changes for the moving average RCLs.

In the MOVAV program registers R13-R32 are the 20-register data block. The sums for the previous 5-, 10-, and 20-days are in registers R10, R11, and R12 respectively. If you write your own moving average program, using the block rotate concept as presented here, then the ROM routine B≥ as well as BR may be used to advantage.

```
01 LBL MOVAV    09 XROM BR     17 RCL 11
02 ST + 10      10 RCL 23      18 10
03 ST + 11      11 ST - 11     19 /
04 ST + 12      12 RCL 18      20 RCL 10
05 X<>32        13 ST - 10     21 5
06 ST - 12      14 RCL 12      22 /
07 13           15 20          23 RTN
08 20           16 /
```

MOVAV will take the datum from X and return the 5-, 10-, and 20-day averages respectively in X, Y, and Z.

Routine Listing For: BR

```
126♦LBL "BR"     141 ST+ Z
127 CHS          142 ST+ Y
128 X<0?         143 DSE L
129 GTO 07       144 GTO 06
130 RCL Y        145 RTN
131 X<>Y         146♦LBL 07
132 1            147 CHS
133 ST+ Z        148 1
134 -            149 -
135 SIGN         150 +
136♦LBL 06       151 STO Y
137 RCL IND Z    152 -1
138 X<> IND Z    153 ST+ Z
139 STO IND T    154 GTO 06
140 RDN
```

LINE BY LINE ANALYSIS OF BR

Line 128 tests for the direction of rotation.

Lines 136-144 are the main loop in the program. The contents of the stack at LBL 06 are determined by the direction of rotation. X holds ±1 and thus either increments or decrements the pointers in Y and Z each time through the loop. The LAST X register is used to hold the DSE counter that controls the loop.

Lines 146-154 are used to intialize the loop.

CONTRIBUTORS HISTORY FOR BR

The BR routine and documentation were written by John Kennedy (918). Martin Sitte (6224) provided the moving average application of BR.

FINAL REMARKS FOR BR

BR is one of the two PPC ROM block routines that does not use the full power of the ISG or DSE control word. (BM is the other) Allowing BR to use the ll portion of the control word will require a new definition of BR for such use.

FURTHER ASSISTANCE ON BR

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 evenings

TECHNICAL DETAILS

XROM: 20, 40 | BR | SIZE: depends on block size

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: ±1
- 4 L: DSE counter

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊦: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

The data registers used are those defined by the block

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine: 06, 07

Execution Time: depends on block size, moves approximately 3.6 registers per second

Peripherals Required: none

Interruptible? yes
Execute Anytime? no
Program File: M2
Bytes In RAM: 51
Registers To Copy: 61

Other Comments:
No special SIZE requirement is necessary provided the data block already exists

BV - BLOCK VIEW

The BV routine provides the 41 user with a fast review of specified data registers. Non-zero data is displayed in the format of R: NNN. Zero data is simply ignored. Each time the data display loop is executed a tone (short 'tic') is executed to let the user know the program is running. This is especially useful when long sequences of non-zero data are encountered. Input for BV is the block definition word bbb.eeeii.

Example 1: Display Registers 0 thru 16.

DO:	SEE:	RESULT:
.016	.016_	bbb.eee
XEQ BV	RN: data	Rapid register review (last register displayed).

COMPLETE INSTRUCTIONS FOR BV

Block View is a classic example of a category of instructions we feel should be in every future programmable RPN calculator. Input for BV is the block definition word bbb.eeeii. With this input BV will display data register contents if the data is not zero. The length of time the display shows a given register value is about 1.2 seconds. If this is not long enough the display time may be doubled by setting flag 9. If the user wants to effectively single step (stop at each non-zero register) through the data, he should set flag 10. Flag 10 is tested before flag 9 and both may be set. This mode does not operate as you would expect, however. Pressing R/S causes the 41 to default to displaying the X register (rather than ALPHA which has the formatted display of register: data). The numeric value is correct, but the flying goose will be displayed during the execution of the routine preparing to display the next non-zero register.

The block control word utilizes the full power of ISG and the ii portion may be effectively used to view every 5th, 3rd, or 10th register etc. If the 82143 A Peripheral Printer is used the display values will be printed if the printer is able to print. BV will not stop if the printer is off. Approximate execution times are given in table 1.

TABLE 1. (1.1 XEQ BV)

REG's	FLAG 9 CLR	FLAG 9 SET	PRINTER* OFF	PRINTER* MAN/NORM
10	11.91	23.59	12.16	13.52
20	23.95	47.79	24.16	27.06
30	35.99	72.26	36.21	40.62
40	48.10	96.78	48.49	54.06
50	60.09	121.40	60.65	67.70
60	72.12	145.84	72.72	81.19
70	84.09	170.38	84.95	94.70
80	96.18	194.96	97.04	108.24
90	108.19	219.41	109.28	121.95
100	120.20	243.89	121.41	135.46

*The printer being plugged-in or not plugged-in doesn't seem to affect the execution time more than 1% (i.e., plugged-in slows execution time less than 1%.)

MORE EXAMPLES OF BV

Example 2: Print the register contents of R00-R26 double wide, fix 3.

DO:	SEE:	RESULT:
SF 12	No Change	Double wide
FIX 3	3 decimal digits	---
.026	.026	Block control word.
XEQ BV	Data	Last value displayed or printed.

Example 3: View Registers 10, 15, 20, 25, & 30.

10.03005 XEQ BV

Example 4: VIEW odd registers 100 thru 200.

101.20002 XEQ BV

Example 5: VIEW (or print) all odd registers of the machine.

SF 25
1.40002
XROM BV

Routine Listing For: BV

```
 99♦LBL "BV"
100 .
101 ENTER↑
102♦LBL 00
103 CLX
104 RCL IND Z
105 X=Y?
106 GTO 01
107 X<> Z
108 INT
109 CLA
110 RCL d
111 CF 29
112 FIX 0
113 ARCL Y
114 STO d
115 "⊢: "
116 R↑
117 ARCL X
118 XROM "VA"
119 FS? 10
120 STOP
121 FS? 09
122 PSE
123 LASTX
124 .
125 ENTER↑
126♦LBL 01
127 TONE 8
128 ISG Z
129 GTO 00
130 TONE 6
131 END
```

LINE BY LINE ANALYSIS OF BV

The routine starts with the block control word in X. Line 100, 101, and 103 clears X, and Y. (The decimal point is faster than zero). The bbb.eeeii value ends up in Z. The display loop starts at line 102. Line 104 recalls the first register. Line 105 compares the recalled value with zero. If zero, lines 126 thru 129 are executed.

Label 01 routine provides the tic, line 127, and increments the block control word and repeats LBL 00 (line 102) or if bbb equals eee return to RAM via line 131.

A non-zero value at line 105 causes LBL 00 routine to continue with line 107. This line places the block control word in X and the previously recalled data in Z. Lines 108 thru 118 format and view the ALPHA register.

T E C H N I C A L   D E T A I L S

XROM: 20,07	BV	SIZE: AS REQUIRED

Stack Usage:	Flag Usage:
0 T: USED	04:
1 Z: USED	05:
2 Y: USED	06:
3 X: USED	07:
4 L: USED	08:
Alpha Register Usage:	09: PAUSES IF SET
5 M: USED	10: STOPS IF SET
6 N: USED	21: Used by VA
7 O: USED	25: Used by VA
8 P: USED	29: USED

Other Status Registers:	Display Mode:
9 Q: NOT USED	As desired.* Excessive digits will scroll and slow execution.
10 ⊢: NOT USED	Angular Mode: N/A
11 a: NOT USED	
12 b: NOT USED	
13 c: NOT USED	
14 d: USED	Unused Subroutine Levels:
15 e: NOT USED	4

ΣREG: NOT USED	Global Labels Called:	
Data Registers:	Direct	Secondary
R00:	VA	NONE
R06: / R07: / R08: As required by data being viewed by user.		
R09:		
R10:		
R11:	Local Labels In This Routine:	
R12:	LBL 00	
	LBL 01	

Execution Time: ZERO DATA: 5.4 Reg. per second.
NON-ZERO DATA: 1.18 seconds per Reg.

Peripherals Required: NONE

		Other Comments:
Interruptible?	YES	* Will print display if printer is able to print and will not stop due to printer being plugged in.
Execute Anytime?	NO	
Program File:	SR	
Bytes In RAM:	59	
Registers To Copy:	40	

Line 108 takes the integer of bbb.eeeii and places bbb.eeeii in last X for future use. ALPHA is cleared in line 109. The flag register is recalled to X in line 110. Lines 111 and 112 makes the current (bbb) register number an integer by clearing flag 29 with FIX 0. Line 113 places this integer in ALPHA. The flags are restored to their pre-routine value at line 114. Line 115 appends a colon and a space for a suitable display. Line 116 rolls up the stack to place the recalled data back into X. Line 117 places this value (displayed in accordance with the display setting) in ALPHA. Line 118 calls the special non-stopping VA routine to view (or print) the ALPHA register. Lines 119 and 120 tests flag 10. If set routine execution stops. If clear (or you press R/S) lines 121 and 122 are executed. If flag 9 is set, a pause is executed for a longer display of the R: NN...N. Line 123 restores the block control word stored at line 108. Lines 124 and 125 restore the stack to the same order as lines 100 and 101. LBL 01 is now executed as described above.

CONTRIBUTORS HISTORY FOR BV

Richard Schwartz (2289) wrote BV for the PPC ROM with inputs and encouragement from the ROM Committe.

FINAL REMARKS FOR BV

Increased speed would be obtained if this routine were in firmware. Also, the stack wouldn't be disturbed and the input block control could be left in X.

FURTHER ASSISTANCE ON BV

Richard J. Nelson (714) 754-6226 P.M.
Richard Schwartz (213) 447-6574 Eve.

NOTES

BX - BLOCK EXTREMA

This routine can be used to find the largest or smallest element of a block of registers. By setting a flag only absolute values of the elements will be considered. The maximum and minimum values as well as their register numbers will be returned. This routine can be used to determine pivot operations in matrix programs.

Example 1: Use BX to find the largest and smallest elements of the following block of registers.

R10: 42
R11: -14
R12: 23
R13: 58
R14: -27
R15: 32
R16: 55
R17: -96
R18: 61
R19: 12
R20: 82
R21: 29
R22: 59
R23: 33

The only input to BX is a block control word which describes the block of data under consideration. In addition, flag 10 controls the absolute value option. Clear flag 10 for this example so that the negative numbers will be considered. For this example key in 10.023 and XEQ "BX". The minimum value is returned in the X-register and the maximum value is returned in the Y-register. The addresses of the minimum and maximum values are returned as the integer parts in the M and N status registers. The output from BX for this example is:

Y: 82 M: 20 = register number of maximum
X: -96 N: 17 = register number of minimum

Example 2: Repeat Example 1 but this time set flag 10 so that absolute values of the numbers are used. All data will now be considered to be positive.

Set F10 and key in 10.023 and XEQ "BX". The following data will be returned.

Y: 96 M: 17 = register of max. absolute value
X: 12 N: 19 = register of min. absolute value

COMPLETE INSTRUCTIONS FOR BX

1) Flag 10 controls an option in which only the absolute values of the numbers will be used to determine the largest and smallest elements of a set of data. If F10 is set then the ABS function is applied to each number before a comparison is made with other data. If F10 is clear then negative numbers may be returned as maxima or minima.

2) The only input to BX is the block control word which describes the block of data under consideration. BX contains an ISG loop and the block control word bbb.eeeii which normally describes any ISG or DSE loop control on the 41C is also used to describe the block of data. BX assumes the block control word is in the X-register when it is called.

3) When BX ends the Y register will contain the largest element found and the X-register will contain the smallest element. (If F10 was set these values will be positive.) The register numbers of these maximum and minimum values will be the integer parts of the M and N status registers. In addition, the O status register will save the block control word. Output from BX routine:

T: *
Z: *
Y: maximum value
X: minimum value

M: maximum register number (INT part)
N: minimum register number (INT part)
O: bbb.eeeii = block control word input

MORE EXAMPLES OF BX

Example 3: Use BX to solve the following two problems. Find the largest element in row 3 and the smallest element in column 4 of the following 6x5 matrix which is assumed to be stored in registers R15-R44.

21	35	55	74	83
11	93	56	36	29
65	78	32	27	75
53	94	46	62	97
54	39	61	67	82
23	45	77	15	25

See Example 1 in the M1 documentation for the exact storage of this matrix. For this example we are primarily concerned with the starting registers of row 3 and column 4.

To find the largest element in row 3 we need to establish the block control word for row 3. Since the matrix rows are stored in consecutive registers and since row 3 starts with register R25 and has 5 elements the desired block control word is simply 25.029. The setting of flag F10 doesn't matter in this example since all the matrix entries are positive. Key in 25.029 and XEQ "BX". The following data is returned.

Y: 78 = maximum value
X: 27 = minimum value

M: 26 (INT part) = max. register
N: 28 (INT part) = min. register
O: 25.029

To find the largest element in column 4 we note that column 4 starts with register R18 and that column entries are separated by the number of columns in the matrix, in this case 5. The last column entry is determined by the number of rows in the matrix. The desired block control word for column 4 is 18.04305. Again the status of flag F10 will not affect the results here since all the matrix elements are positive so simply key 18.04305 and XEQ "BX". The following data are returned:

Y: 74 = maximum value
X: 15 = minimum value

M: 18 (INT part) = max. register
N: 43 (INT part) = min. register
O: 18.04305

APPLICATION PROGRAM 1 FOR BX

See the RRM program in the M1 routine documentation.

Routine Listing For: **BX**

```
155♦LBL "BX"        175 RDN
156 STO [           176♦LBL 09
157 STO \           177 ISG Z
158 STO ]           178 GTO 08
159 RCL IND X       179 X<>Y
160 FS? 10          180 R↑
161 ABS             181 RTN
162 ENTER↑          182♦LBL 10
163 ENTER↑          183 X<>Y
164 RDN             184 CLX
165♦LBL 08          185 RCL Z
166 CLX             186 STO [
167 RCL IND Z       187 GTO 09
168 FS? 10          188♦LBL 11
169 ABS             189 CLX
170 X>Y?            190 RCL T
171 GTO 10          191 STO \
172 R↑              192 X<>Y
173 X>Y?            193 RDN
174 GTO 11          194 GTO 09
```

LINE BY LINE ANALYSIS OF BX

Lines 156-158 store the block control word in registers M, N, and O. As a new maximum or minimum is found registers M and N will change but the original block control word is preserved in O.

Lines 159-164 store the first element of the block in Y and T. This initializes the stack with the maximum and minimum values in Y and T respectively.

Lines 165-181 are the main loop in the program. At LBL 08, line 165 the stack contains:

X: scratch Y: max Z: block control word T: min

The next element from the block is recalled and compared with the current max/min. Line 170 tests for a new maximum and transfers control to LBL 10 if a new maximum is found. Line 173 tests for a new minimum and transfers control to LBL 11 if a new minimum is found. LBL 09 serves as the continuation point from processing a new max/min after a transfer to LBL 10 or LBL 11 whose purpose is to update the new addresses and values of the new max/min point.

CONTRIBUTORS HISTORY FOR BX

The BX routine is by Richard Schwartz (2289). John Kennedy (918) provided the documentation for BX.

FURTHER ASSISTANCE ON BX

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

TECHNICAL DETAILS

XROM: 20, 41 — **BX** — SIZE: depends on block size

Stack Usage:	Flag Usage:
0 T: used	04: not used
1 Z: used	05: not used
2 Y: used	06: not used
3 X: used	07: not used
4 L: used	08: not used
Alpha Register Usage:	09: not used
5 M: max address INT	10: used for ABS option
6 N: min address INT	
7 O: block word	
8 P: not used	25: not used
Other Status Registers:	**Display Mode:**
9 Q: not used	not used
10 ⊢: not used	
11 a: not used	**Angular Mode:**
12 b: not used	not used
13 c: not used	
14 d: not used	**Unused Subroutine Levels:**
15 e: not used	5
ΣREG: not used	**Global Labels Called:** Direct: none; Secondary: none
Data Registers: R00: ; R06: ; R07: ; R08: ; R09: ; R10: ; R11: ; R12: — The data registers used depend on the block used	**Local Labels In This Routine:** 08, 09, 10, 11

Execution Time: data dependent; approximate range 2.7 - 3.6 registers per second; nominal time for 100 registers is 30 seconds

Peripherals Required: none

Interruptible? yes	**Other Comments:**
Execute Anytime? no	No special SIZE requirement is necessary provided the data blocks already exist
Program File: M2	
Bytes In RAM: 65	
Registers To Copy: 61	

BΣ - BLOCK STATISTICS

This routine is called Block Statistics but may be considered part of the matrix group since it is designed to compute vector dot products. Vector dot products are the internal operations required to compute matrix products. This routine can also be used with other than matrix elements to compute sums and sums of squares and cross products of blocks of data.

Example 1: The following two blocks of data registers contain the data indicated. Use BΣ to compute the sum of the products of corresponding data elements. If the data are considered to be rows or columns of a matrix then the sum of the products of corresponding elements is the dot product of the vectors represented by the rows or columns.

R20: 5	R30: -4
R21: -3	R31: 6
R22: 4	R32: 7
R23: 6	R33: -8
R24: -9	R34: 2
R25: 8	R35: 5
R26: 2	R36: 9
R27: -1	R37: 7
R28: 5	R38: -6
R29: -7	R39: 1

First store the data in the indicated registers. Because BΣ uses the Σ+ function the six Σ registers must be assigned to registers which will not conflict with other data. Key ΣREG 10 to assign the Σ registers to registers R10-R15 inclusive. The Σ registers do not need to be cleared before BΣ is called since CLΣ is the first instruction that BΣ executes.

The input to BΣ is the two block control words for the two blocks of data whose statistics we wish to compute. Since in this example both blocks consist of consecutive blocks of registers we simply key in 20.029 ENTER↑ 30.039 and XEQ " BΣ ". The results listed below are left in the statistical registers. The numerical values are from the data of the example.

R10: Σx = sum of 30.039 block elements = 19

R11: Σx^2 = sum of squares of 30.039 block = 361

R12: Σy = sum of 20.029 block elements = 10

R13: Σy^2 = sum of squares of 20.029 block = 310

R14: Σxy = sum of cross products = vector dot product = -62

R15: n = count of number of iterations, depends on

the block control word for the first block entered = 10.

The answer of the vector dot product for this example is -62. The sums of the two blocks and the sums of the squares from the two blocks are also left in the statistical registers.

COMPLETE INSTRUCTIONS FOR BΣ

1) BΣ assumes the data elements of the two blocks are stored in the registers which will be described (pointed to) by the two block control words that are the only input to BΣ .

2) The assignment of the location of the statistics registers should be considered before BΣ is called. The two blocks will be preserved as long as the assignment of the statistics registers does not overlap on either block.

3) The standard form of the block control words is bbb.eeell which conforms to the block control words the 41C uses for the ISG and DSE functions. These two block control words should be in the Y and X registers when BΣ is called. For most applications the two blocks of data will be the same length and the order of the two block control words input is unimportant. However, internally BΣ uses an ISG loop and tests only the block control word that was entered in the Y-register to determine the end of the data. BΣ does not preserve any of the stack registers.

4) XEQ " BΣ " and the following data will be left in the statistics registers.

Σx = sum of X-register block

Σx^2 = sum of squares of X-register block

Σy = sum of Y-register block

Σy^2 = sum of squares of Y-register block

Σxy = sum of cross products of the two blocks

n = number of iterations on Y-register block

MORE EXAMPLES OF BΣ

Example 2: Use BΣ to accumulate statistical sums for every 3rd register in the first block shown below with every other register in the second block shown below.

1st block	2nd block
R15: 7	R31: -5
R16: 3	R32: 9
R17: 14	R33: 20
R18: -7	R34: 8
R19: 13	R35: -16
R20: 15	R36: 29
R21: 8	R37: 32
R22: 24	R38: 12
R23: 40	R39: 32
R24: -12	R40: -7
R25: 21	R41: 12
R26: 5	
R27: 34	
R28: 30	
R29: -3	
R30: 15	

The block control word for the first block will be 15.03003 and for the second block the the control word will be 31.04102. Assign the Σ registers starting at R45. Key ΣREG 45. Then key

15.03003 ENTER↑ 31.04102 and XEQ " BΣ "

The following sums will be accumulated starting at R45.

R45: 75	R48: 1687
R46: 2873	R49: 581
R47: 45	R50: 6

The pairings between the two blocks based on the control words used in this example are:

R15 & R31 = first pair
R18 & R33 = second pair
R21 & R35 = third pair
R24 & R37 = fourth pair
R27 & R39 = fifth pair
R30 & R41 = sixth and last pair

Example 3: Apply BΣ to the following block of data where the two block control words are the same and both apply to this single block.

R15: 4	R21: 6
R16: 5	R22: -5
R17: 3	R23: 6
R18: 9	R24: 13
R19: 8	R25: 12
R20: -4	R26: 7

Assign the Σ registers starting at R30. Key ΣREG 30. The block control word is simply 15.026. Key 15.026 ENTER↑ and XEQ " BΣ ".

The following data is left in the statistical registers. Note the multiple copies of Σx and also Σx^2.

R30:	$\Sigma x = 64$	same as R32	
R31:	$\Sigma x^2 = 670$	same as R33 & R34	
R32:	$\Sigma y = 64$	same as R30	
R33:	$\Sigma y^2 = 670$	same as R31 & R34	
R34:	$\Sigma xy = 670$	same as R31 & R33	
R35:	$n = 12$		

Routine Listing For: BΣ

```
195♦LBL "BΣ"
196 CLΣ
197♦LBL 12
198 RCL IND Y
199 RCL IND Y
200 Σ+
201 R↑
202 R↑
203 ISG X
204 ""
205 ISG Y
206 GTO 12
207 RTN
```

LINE BY LINE ANALYSIS OF BΣ

Line 196 clears the statistics registers before BΣ starts. The two block control words are assumed to be in X and Y.

Lines 197-206 are the main loop in the routine. Corresponding elements from each block are recalled in X and Y and the the Σ + function is applied.

CONTRIBUTORS HISTORY FOR BΣ

The BΣ routine and documentation are by John Kennedy (918) with help from Richard Schwartz (2289).

FURTHER ASSISTANCE ON BΣ

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

TECHNICAL DETAILS

XROM: 20, 42 — BΣ — SIZE: depends on blocks used

Stack Usage:
0 T: used
1 Z: used
2 Y: used
3 X: used
4 L: used

Flag Usage:
04: not used
05: not used
06: not used
07: not used
08: not used
09: not used
10: not used
25: not used

Alpha Register Usage:
5 M: not used
6 N: not used
7 O: not used
8 P: not used

Other Status Registers:
9 Q: not used
10 ⊢: not used
11 a: not used
12 b: not used
13 c: not used
14 d: not used
15 e: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 5

ΣREG: used

Data Registers:
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

The data registers depend on the two blocks and the location of the Σ registers

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine: 12

Execution Time: depends on block size; accumulates approximately 2.0 register pairs per second

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: M2

Bytes In RAM: 25

Registers To Copy: 61

Other Comments:
No special SIZE requirement is necessary provided the data blocks already exist

C? - CURTAIN FINDER

C? provides the location of the "curtain" separating data and program memory to S? , EP , and other PPC ROM routines. S? uses this information to determine the SIZE modulo 64, while EP uses it to determine what registers to clear. C? is most useful in connection with other heavily synthetic routines.

Example 1: The curtain location changes with the SIZE. For instance:

DO:	SEE:	RESULT:
SIZE 010		
XEQ C?	502	Curtain location. May be 246, 310, 374, 430, or 502, depending on RAM complement.
SIZE 008		
XEQ C?	504	Curtain is now two registers higher, reducing the number of data registers by two.

COMPLETE INSTRUCTIONS FOR C?

XEQ C? to place in X a decimal number indicating the absolute address of the curtain in program memory. The former contents of X and Y are preserved in Y and Z.

C? can be used to follow the behavior of curtain control programs (see the appendix on curtain moving). If you XEQ C? before moving the curtain and write down the result, you can always recover the original curtain position by entering the same number and pressing XEQ CX .

LINE BY LINE ANALYSIS OF C?

46 LBL C?	
47 RCL c	place register c into X - 1st 3 nybbles = ΣREG Location
48 LBL 14	9, 10, 11 nybbles = curtain address 12, 13, 14 nybbles = permanent END
49 STO M	place register c into M
50 Hex F4 7F 00 00 41	append two nulls and the letter A (hex 41)
51 X<>Y	X = last 4 bytes of c + 2 nulls +A
52 X<>d	exchange with d; the letter A sets flags 49 and 55 to avoid problems with interruption
53 CF 01 54 CF 02 55 CF 04 56 CF 07	Clear the 69 portion of the cold start constant normally in nybbles 7 and 8 of register c. The curtain address is now located in hex in flags 8-19 of register d. Flags 8-9 are always clear.
57 FS?C 10 58 SF 07 59 FS?C 11 60 SF 09 61 FS?C 12 62 SF 10 63 FS?C 13 64 SF 11 65 FS?C 14 66 SF 13 67 FS?C 15 68 SF 14 69 FS?C 16 70 SF 15	Convert the hex number represented by flags 10-19 into an octal number represented by flags 7-19. The status of flag 10 is shifted left three bits, the status of flags 11, 12, and 13 are shifted left two bits, and the status of flags 14, 15, and 16 is shifted left one bit.

This procedure, pioneered by Roger Hill (4940), set up octal-decimal conversion later in this program.

71 X<>d	restore previous d, places modified curtain address into X
72 E38	
73 /	x = curtain address in numeral format, exponent 3
74 INT	truncate the fractional portion of x containing the .END. location
75 DEC	convert the octal number in x to decimal
76 RTN	

C?

Routine Listing For: C?

```
46♦LBL "C?"        61 FS?C 12
47 RCL c           62 SF 10
                   63 FS?C 13
48♦LBL 14          64 SF 11
49 STO [           65 FS?C 14
50 "├♦♦A"          66 SF 13
51 X<> [           67 FS?C 15
52 X<> d           68 SF 14
53 CF 01           69 FS?C 16
54 CF 02           70 SF 15
55 CF 04           71 X<> d
56 CF 07           72 E38
57 FS?C 10         73 /
58 SF 07           74 INT
59 FS?C 11         75 DEC
60 SF 09           76 RTN
```

CONTRIBUTORS HISTORY FOR C?

The need for a curtain finder became apparent with the advent of synthetic SIZE finders (see *PPC CALCULATOR JOURNAL*, V7N5P57a and 57d). Other uses appeared soon thereafter. This version of C? was written by Clifford Stern (4516) as part of the S? / ΣΣ? / C? package.

FURTHER ASSISTANCE ON C?

Call Clifford Stern (4516) at (213) 748-0706.

Call Keith Jarett (4360) at (213) 374-2583.

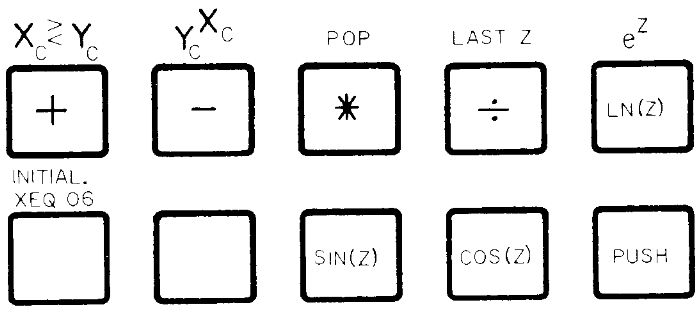
TECHNICAL DETAILS		
XROM: 10,16	**C?**	SIZE: 000

Stack Usage: 0 T: Y 1 Z: Y 2 Y: X 3 X: curtain 4 L: octal curtain	Flag Usage: MANY USED BUT ALL RESTORED 04: 05: 06: 07: 08:
Alpha Register Usage: 5 M: 6 N: ALL USED 7 O: 8 P:	09: 10: 25:
Other Status Registers: 9 Q: NOT USED 10 ⊢: NOT USED 11 a: NOT USED 12 b: NOT USED 13 c: NOT USED 14 d: USED BUT RESTORED 15 e: NOT USED	Display Mode: UNCHANGED Angular Mode: UNCHANGED Unused Subroutine Levels: 6
ΣREG: UNCHANGED Data Registers: NONE USED R00: R06: R07: R08: R09: R10: R11: R12:	Global Labels Called: Direct: NONE Secondary: NONE Local Labels In This Routine: 14
Execution Time: 1.1 second.	
Peripherals Required: NONE	
Interruptible? YES Execute Anytime? YES Program File: ML Bytes In RAM: 65 Registers To Copy: 64	Other Comments:

NOTES

CA - COMPLEX ARITHMETIC

The complex arithmetic program provides some basic complex arithmetic operations. These operations are performed on complex numbers which are considered to be elements of a complex stack. To avoid confusion with the HP built-in 4 level RPN XYZT stack, in the following discussion the words "complex stack" will refer to the artificial complex number stack simulated in the data registers. The complex number functions provided are: addition, subtraction, multiplication, division, natural log and anti-log, complex Y to the power of complex X, complex sine and complex cosine. The complex stack functions are: complex X exchange Y, a complex Last X function called Last Z, and complex stack Push and Pop operations. The keyboard arrangement of the functions is indicated in the diagram below. These are the top two rows of keys.



BACKGROUND FOR CA

Complex Stack Discussion

A complete understanding of the workings of the complex stack is required to be able to effectively use the functions provided. In fact, understanding the complex stack is essentially all there is to understanding the entire complex arithmetic program. The following discussion assumes complete familiarity with HP's 4 level RPN stack. The operation of the complex stack is simple and is similar to the regular XYZT stack but there are some differences.

The major difference is that the complex stack does not have a fixed size of 4 elements. The only limitation on the length of the complex stack is the number of available data registers. The first complex number entered on the complex stack is considered to be the top of the stack. The bottom two elements on the complex stack are called complex Y and complex X, complex X being on the bottom of the stack. The size of the complex stack dynamically expands and contracts as numbers are added to the stack and operations are performed. The expansion or contraction takes place at the bottom of the complex stack.

Each complex number may be considered as a pair of real numbers, one of which is called the real part of the complex number, the other is called the imaginary part of the complex number. We will write complex numbers in the form X + iY where X is the real part and Y is the imaginary part. Each complex number on the complex stack uses two data registers. The first or top element on the complex stack occupies R10 and R11, with R10 holding the imaginary part (Y) and R11 holding the real part (X). The next complex number will occupy R12 and R13, the next R14 and R15, etc. Successive complex numbers occupy two more registers and are always stored with the imaginary part in the even numbered register and the real part in the odd numbered register.

The Last Z function operates on the complex stack exactly as does Last X on the XYZT stack. When a one- or two-number operation is performed, the bottom element on the complex stack is first saved in Last Z. The actual location of Last Z is in R07 and R08, with the imaginary part in R07 and the real part in R08.

R09 acts as a bottom of stack pointer and is used to keep track of the current length of the complex stack. Essentially R09 holds only the odd numbers 11, 13, 15, ... Thus if R09 = 19 there will be 5 complex numbers on the complex stack; complex X occupying R18 & R19, complex Y occupying R16 & R17. The top of the complex stack is always in R10 & R11 as in the diagram below. The complex stack is an RPN LIFO (last in first out) FORTH-like type stack.

Diagram of Complex Stack With R09 = 19

R07: R08:	>Last Z
R09:	bottom of stack pointer = 19 in this example
R10: R11:	>1st number pushed on stack = top of stack
R12: R13:	>2nd number pushed on stack
R14: R15:	>3rd number pushed on stack
R16: R17:	>4th number = complex Y
R18: R19:	>5th number = complex X = bottom of stack

The one-number operations e^z, Ln(z), sin(z), and cos(z) operate on the complex number on the bottom of the complex stack (complex X) and leave their results in the space occupied by complex X. One-number operations do not change the length of the complex stack.

The two-number operations +, -, *, /, y^x operate on the two bottom elements of the complex stack (complex Y and complex X), leave their answers in the space occupied by complex Y, and thus contract the length of the complex stack by one complex number.

The Push and Pop functions interface the XYZT stack with the complex stack. Push expands the complex stack by appending the number X + iY (X & Y are from the XYZT stack) onto the bottom of the complex stack. Pop is the opposite of Push. Pop contracts the complex stack by removing the bottom element and placing it in the XYZT stack where the number is assumed to be in the form X + iY.

Pop is really an internal subtroutine. The Pop function will probably be used infrequently because all the one- and two-number operations leave their results on both stacks. The results are naturally left on the complex stack so that successive operations can be chained, but the results are also left in X and Y as a matter of convenience for the user to view using the regular X<>Y or R↓ keys. In other words, it isn't necessary to deliberately Pop the complex stack to see the result of an operation.

In contrast, the Push function must be used to place any and all complex numbers on the complex stack before any operations can be performed. Push is the complex analog of ENTER↑ but only one copy of the complex number is pushed onto the complex stack and

CA

there is no need for stack enable/disable. Push will be the most used of all the routines in the CA program.

The complex X exchange Y is the analog of the regular X<>Y and simply reverses the positions of the bottom two complex numbers on the complex stack. The length of the complex stack does not change in this case.

Last Z has previously been mentioned as the complex analog of Last X. Last Z recalls the complex number from R07 & R08 and Pushes it onto the bottom of the complex stack. The complex stack expands each time Last Z is used.

Finally we come to the operation of initializing the complex stack. Performing XEQ 06 is the complex stack initialization. This routine stores the number 9.009 in R09 so that after the first complex number is Pushed onto the complex stack R09 will be pointing to R11. R09 is increased by 2 after a Push operation.

The purpose of the decimal .009 is to avoid complex stack underflow and provide a means of simulating the replication of the T-register. Since the complex stack does not have a fixed length the usual T-register dropping action of the XYZT stack has not been duplicated in the complex stack. Instead, the Pop operation functions as if many copies of the top element on the complex stack were available. Only one real copy exists in R10 & R11. Popping the complex stack when there is only the top element on the complex stack does not remove that element from the complex stack (although a copy is left in the XYZT stack in X & Y). R09 remains pointing to R11.

When there are only two complex numbers on the complex stack and a two-number operation is performed, the top element remains as the top element on the complex stack and the correct result is stored as the second element on the complex stack. This is the only exception to the rule that two-number operations result in a net decrease in the length of the complex stack by one complex number. Examples of the use of this replication feature will be given later on. The procedure to remove (expunge) the top element from the complex stack is simply to re-initialize the stack with XEQ 06.

XEQ 06 can be thought of as clearing the complex stack although no registers are actually zeroed out. Just as with the XYZT stack, it isn't necessary to clear the complex stack before beginning a new operation, but pushing more elements onto the complex stack will increase its size. This is in contrast to the XYZT stack where elements may be lost if pushed off the top of the stack. Users will find it a practical matter to periodically clear the complex stack by XEQ 06.

<u>Example 1:</u> Compute (2+3i)*[(7-6i)+(4+5i)]

Plug the PPC ROM into the 41C and SIZE 020 minimum. GTO "CA". This last step insures that the complex functions will be assigned to the top two rows of keys. Square brackets [ ] will be used to enclose the functions on the keys. The instruction push [xxx] simply means push the key marked with the function xxx. The results in this first example are shown in FIX 0 display mode.

XEQ 06 to initialize the stack. In this example the first complex number to be pushed onto the complex stack is 2+3i. Key 3 ENTER↑ 2 [PUSH]. The complex stack now appears as:

```
Y: 3i        R09: 11 = bottom of stack pointer
X: 2         R10: 3i
             R11: 2  = bottom of complex stack
```

(the i character is not present in the registers)

The second complex number to be entered is 7-6i. Key 6 CHS ENTER↑ 7 [PUSH]. The complex stack now appears as:

```
Y: -6i       R09: 13 = bottom of stack pointer
X: 7         R10: 3i
             R11: 2
             R12: -6i
             R13: 7  = bottom of complex stack
```

The third number to be entered is 4+5i. Key 5 ENTER↑ 4 [PUSH]. The complex stack now contains:

```
Y: 5i        R09: 15 = bottom of stack pointer
X: 4         R10: 3i
             R11: 2
             R12: -6i
             R13: 7
             R14: 5i
             R15: 4  = bottom of complex stack
```

Now press [+] to add the two complex numbers within the square brackets. The status of the machine is indicated by:

```
Y: -i        R07: 5i
X: 11        R08: 4   >Last Z
             R09: 13 = bottom of stack pointer
             R10: 3i
             R11: 2
             R12: -i
             R13: 11 = bottom of complex stack
```

Finally press [*] to perform the multiplication. The final machine status is indicated by:

```
Y: 31i       R07: -i
X: 25        R08: 11  >Last Z
             R09: 13 = bottom of stack pointer
             R10: 3i
             R11: 2
             R12: 31i
             R13: 25 = bottom of complex stack
```

Note that intermediate results and final results appear in X and Y as well as on the bottom of the complex stack ready for chaining of further operations. These results may be viewed at any time in the X & Y registers with a simple X<>Y or R↓.

COMPLETE INSTRUCTIONS FOR CA

(Keyboard Operations)

These instructions assume a knowledge of the previous background discussion of the complex stack operations.

1) Key GTO "CA" and set a minimum size at least as large as SIZE n where n=10+2j and j is the maximum size stack desired. (For example, if j=5, SIZE 020 will allow a 5 high stack). The keyboard functions should now be available on the top two rows of keys.

2) XEQ 06 to initialize the stack. The bottom of the stack is pointed to by R09 which is incremented or

decremented by 2 each time a complex pair is added or removed from the complex stack.*

3) All complex pairs are assumed to be in rectangular form as X + iY and must be PUSHED onto the complex stack from X & Y in the XYZT stack. Key Y ENTER↑ X and [PUSH].

4) The one-number functions e^z, Ln(z), sin(z), and cos(z) all take their arguments from the bottom of the complex stack. Use radians mode for these functions. These functions leave their results on the bottom of the complex stack as well as in X & Y in the XYZT stack. These functions save Z in Last Z. (For example, if z = 2+3i, to compute e^z key 3 ENTER↑ 2 [PUSH] [e^z] See 7.3789).

5) The two-number functions +, -, *, /, and y^x take their arguments from the last two positions on the bottom of the complex stack. The power function should be used in radians mode. The bottom-most element is saved in Last Z except that the power function does not save Last Z. These two-number functions leave their results in X & Y in the XYZT stack as well as on the new bottom of the complex stack which now holds one less complex number than when the two-number function was called. (For example, to multiply (4+5i)*(2-7i) key 5 ENTER↑ 4 [PUSH] 7 CHS ENTER↑ 2 [PUSH] [*] See 43.0000).

6) The Last Z function PUSHES the number stored in the Last Z registers (R07 & R08) onto the complex stack.

7) The complex X exchange Y function exchanges the bottom two complex numbers on the complex stack.

8) The Pop stack function removes the bottom number from the complex stack, places this number in X & Y in the XYZT stack, and decreases the complex stack pointer by 2.*

* When a complex number is removed from the complex stack via a Pop operation the stack pointer R09 is decreased by 2 as long as more than one complex number is on the complex stack. If only one number is on the complex stack then the Pop operation leaves that number on the complex stack and R09 is not decremented. This feature allows the top stack element to be replicated, but may be defeated to allow a new top element by re-initializing the stack (XEQ 06).

MORE EXAMPLES OF CA

For these examples use a minimum size of SIZE 020 and insure the calculator is in RADIANS angle mode. GTO "CA" so the program pointer is in ROM and the complex functions are on the top rows of keys. Square brackets will be used to enclose the names of functions on the keys.

Example 2: Use CA to calculate the following formula:

$$(3-5i)*(4+2i) + (6+2i)*(1-4i) + (3+2i)^2$$

For this example we will show a complete stack trace. These results are shown in FIX 0 display mode.

XEQ 06 to initialize the stack. (3-5i) is the first complex number to be pushed onto the complex stack. Key 5 CHS ENTER↑ 3 [PUSH]. The stack contents are:

Y: -5	R09: 11 = bottom of stack pointer
X: 3	R10: -5
	R11: 3

Next key 2 ENTER↑ 4 [PUSH]. The stack changes to:

Y: 2	R09: 13 = bottom of stack pointer
X: 4	R10: -5
	R11: 3
	R12: 2
	R13: 4

Now multiply [*]. The stack changes to:

Y: -14	R09: 13 = bottom of stack pointer
X: 22	R10: -5
	R11: 3
	R12: -14
	R13: 22

Next key 2 ENTER↑ 6 [PUSH]. The stack becomes:

Y: 2	R09: 15 = bottom of stack pointer
X: 6	R10: -5
	R11: 3
	R12: -14
	R13: 22
	R14: 2
	R15: 6

The multiplication cannot be performed yet so continue by keying in the next number. Key 4 CHS ENTER↑ 1 [PUSH]. The stack becomes:

Y: -4	R09: 17 = bottom of stack pointer
X: 1	R10: -5
	R11: 3
	R12: -14
	R13: 22
	R14: 2
	R15: 6
	R16: -4
	R17 1

Now multiply. Key [*].

Y: -22	R09: 15 = bottom of stack pointer
X: 14	R10: -5
	R11: 3
	R12: -14
	R13: 22
	R14: -22
	R15: 14

Then add the two products. [+].

Y: -36	R09: 13 = bottom of stack pointer
X: 36	R10: -5
	R11: 3
	R12: -36
	R13: 36

Finally key in the number to be squared. Key 2 ENTER↑ 3 [PUSH].

Y: 2	R09: 15 = bottom of stack pointer
X: 3	R10: -5
	R11: 3
	R12: -36
	R13: 36
	R14: 2
	R15: 3

Key in the exponent 2 as the complex number 2+0i. Key 0 ENTER↑ 2 [PUSH].

```
Y: 0          R09: 17 = bottom of stack pointer
X: 2          R10: -5
              R11: 3
              R12: -36
              R13: 36
              R14: 2
              R15: 3
              R16: 0
              R17: 2
```

Then key [Y↑X] to perform the squaring operation.

```
Y: 12         R09: 15 = bottom of stack pointer
X: 5          R10: -5
              R11: 3
              R12: -36
              R13: 36
              R14: 12
              R15: 5
```

Finally add the last term to obtain the final answer. [+].

```
Y:-24         R09: 13 = bottom of stack pointer
X: 41         R10: -5
              R11: 3
              R12: -24
              R13: 41
```

The final answer is 41 - 24i.

Example 3: This example will illustrate the use of some of the other complex functions. All results are shown to four decimal places. Use RADIANS mode.

Calculate sin(2+3i) + cos(1-4i) + e^{2+2i}

XEQ 06 to initialize the stack and key in the first number as 3 ENTER↑ 2 [PUSH].

```
Y: 3.0000     R09: 11.0090 = bottom of stack pointer
X: 2.0000     R10: 3.0000
              R11: 2.0000
```

Then compute the sine by keying [sin(z)]. Note how the top stack element is saved on top of the stack.

```
Y:-4.1689     R09: 13.0090 = bottom of stack pointer
X: 9.1545     R10: 3.0000
              R11: 2.0000
              R12: -4.1689
              R13: 9.1545
```

Next key in the argument for the cosine as 4 CHS ENTER↑ 1 [PUSH].

```
Y:-4.0000     R09: 15.0090 = bottom of stack pointer
X: 1.0000     R10: 3.0000
              R11: 2.0000
              R12: -4.1689
              R13: 9.1545
              R14: -4.0000
              R15: 1.0000
```

Then calculate the cosine. [cos(z)].

```
Y: 22.9637    R09: 15.0090 = bottom of stack pointer
X: 14.7547    R10: 3.0000
              R11: 2.0000
              R12: -4.1689
              R13: 9.1545
              R14: 22.9637
              R15: 14.7547
```

Then add the sine and cosine terms. [+].

```
Y: 18.7948    R09: 13.0090 = bottom of stack pointer
X: 23.9092    R10: 3.0000
              R11: 2.0000
              R12: 18.7948
              R13: 23.9092
```

Now key 2 ENTER↑ [PUSH] to enter the argument for the exponential term.

```
Y: 2.0000     R09: 15.0090 = bottom of stack pointer
X: 2.0000     R10: 3.0000
              R11: 2.0000
              R12: 18.7948
              R13: 23.9092
              R14: 2.0000
              R15: 2.0000
```

Calculate the exponential term by keying [e^z].

```
Y: 6.7188     R09: 15.0090 = bottom of stack pointer
X:-3.0749     R10: 3.0000
              R11: 2.0000
              R12: 18.7948
              R13: 23.9092
              R14: 6.7188
              R15: -3.0749
```

Then add the last two numbers by keying [+] and see the final stack contents as:

```
Y: 25.5136    R09: 13.0090 = bottom of stack pointer
X: 20.8343    R10: 3.0000
              R11: 2.0000
              R12: 25.5136
              R13: 20.8343
```

Example 4: Calculate ln(4+5i) + $(3-2i)^{(2-3i)}$

XEQ 06 to initialize the stack and key in the first entry as 5 ENTER↑ 4 [PUSH].

```
Y: 5.0000     R09: 11 = bottom of stack pointer
X: 4.0000     R10: 5.0000
              R11: 4.0000
```

Then calculate the natural log. [ln(z)].

```
Y: 0.8961     R09: 13 = bottom of stack pointer
X: 1.8568     R10: 5.0000
              R11: 4.0000
              R12: 0.8961
              R13: 1.8568
```

Then key in the base of the power term as 2 CHS ENTER↑ 3 [PUSH].

```
Y:-2.0000     R09: 15 = bottom of stack pointer
X: 3.0000     R10: 5.0000
              R11: 4.0000
              R12: 0.8961
              R13: 1.8568
              R14: -2.0000
              R15: 3.0000
```

Then key in the exponent as 3 CHS ENTER↑ 2 [PUSH].

Y:-3.0000   R09: 17 = bottom of stack pointer
X: 2.0000   R10: 5.0000
R11: 4.0000
R12: 0.8961
R13: 1.8568
R14: -2.0000
R15: 3.0000
R16: -3.0000
R17: 2.0000

Then calculate the power. [Y↑X].

Y: 2.1207   R09: 15 = bottom of stack pointer
X: 0.6818   R10: 5.0000
R11: 4.0000
R12: 0.8961
R13: 1.8568
R14: 2.1207
R15: 0.6818

And then perform the addition. [+].

Y: 3.0168   R09: 13 = bottom of stack pointer
X: 2.5386   R10: 5.0000
R11: 4.0000
R12: 3.0168
R13: 2.5386

Note that the intermediate results were left on the bottom of the complex stack as well as in Y and X.

Example 5: The stack replication feature will be illustrated by using an example of polynomial evaluation. The user is assumed to be familiar with the method known as Horner's nesting algorithm which is an efficient method for evaluating a polynomial which is not to be confused with synthetic division. This example could also be given for a real-valued polynomial using the XYZT stack.

Evaluate $p(z) = 9z^4 + 5z^3 - 4z^2 + 3z - 5$ at z=(2+3i).

The idea behind Horner's scheme is to have several copies of the argument available. Then starting with the leading coefficient, multiply with the argument and then add the next coefficient. Then multiply again with the argument and add the next coefficient and repeat these steps until the last coefficient has been added. For this example z is complex but all the coefficients are real. The results shown here are in FIX 0 display mode.

XEQ 06 to initialize the stack and then key in the value z as 3 ENTER↑ 2 [PUSH].

Y: 3   R09: 11 = bottom of stack pointer
X: 2   R10: 3
R11: 2

Key in the leading coefficient as 0 ENTER↑ 9 [PUSH].

Y: 0   R09: 13 = bottom of stack pointer
X: 9   R10: 3
R11: 2
R12: 0
R13: 9

Then multiply. [*].

Y: 27   R09: 13 = bottom of stack pointer
X: 18   R10: 3
R11: 2
R12: 27
R13: 18

Notice that the argument z remains on top of the stack. Now enter the next coefficient. 0 ENTER↑ 5 [PUSH].

Y: 0   R09: 15 = bottom of stack pointer
X: 5   R10: 3
R11: 2
R12: 27
R13: 18
R14: 0
R15: 5

Add the coefficient. [+].

Y: 27   R09: 13 = bottom of stack pointer
X: 23   R10: 3
R11: 2
R12: 27
R13: 23

Now multiply. [*].

Y: 123   R09: 13 = bottom of stack pointer
X: -35   R10: 3
R11: 2
R12: 123
R13: -35

Then key in the next coefficient. 0 ENTER↑ 4 CHS [PUSH].

Y: 0   R09: 15 = bottom of stack pointer
X: -4   R10: 3
R11: 2
R12: 123
R13: -35
R14: 0
R15: -4

Then add. [+]

Y: 123   R09: 13 = bottom of stack pointer
X: -39   R10: 3
R11: 2
R12: 123
R13: -39

Then multiply. [*].

Y: 129   R09: 13 = bottom of stack pointer
X: -447   R10: 3
R11: 2
R12: 129
R13: -447

Key in the next coefficient as 0 ENTER↑ 3 [PUSH].

Y: 0   R09: 15 = bottom of stack pointer
X: 3   R10: 3
R11: 2
R12: 129
R13: -447
R14: 0
R15: 3

Then add. [+].

Y: 129   R09: 13 = bottom of stack pointer
X: -444   R10: 3

```
R11: 2
R12: 129
R13: -444
```

Then multiply. [*].

```
Y: -1074      R09: 13 = bottom of stack pointer
X: -1275      R10: 3
              R11: 2
              R12: 129
              R13: -447
```

Finally key in the last coefficient. 0 ENTER↑ 5 CHS [PUSH].

```
Y: 0          R09: 15 = bottom of stack pointer
X: -5         R10: 3
              R11: 2
              R12: -1074
              R13: -1275
              R14: 0
              R15: -5
```

Add the last coeffient to obtain the final answer. [+].

```
Y: -1074      R09: 13 = bottom of stack pointer
X: -1280      R10: 3
              R11: 2
              R12: -1074
              R13: -1280
```

FORMULAS USED IN CA

Let x, y, x_1, y_1, x_2, y_2 denote real numbers.

Let z, z_1, z_2 denote complex numbers where:

$$z = x + i*y$$

$$z_1 = x_1 + i*y_1$$

$$z_2 = x_2 + i*y_2$$

(1) $z_1 + z_2 = (x_1+x_2) + i*(y_1+y_2)$

(2) $z_1 - z_2 = (x_1-x_2) + i*(y_1-y_2)$

(3) $z_1 * z_2 = (x_1x_2 - y_1y_2) + i*(x_1y_2 + x_2y_1)$

(4) $z_1 / z_2 = (x_1x_2+y_1y_2)/(x_2^2+y_2^2) + i*[x_2y_1-x_1y_2)/(x_2^2+y_2^2)]$

(5) $\ln(z) = \ln(\sqrt{x^2 + y^2}) + i*(\tan^{-1}(y/x))$

(6) $e^z = e^x\sin(y) + i*e^x\cos(y)$

(7) $\sin(z) = \sin(x)\cosh(y) + i*\cos(x)\sinh(y)$

(8) $\cos(z) = \cos(x)\cosh(y) - i*\sin(x)\sinh(y)$

(9) $\cosh(y) = (e^y + e^{-y})/2$

(10) $\sinh(y) = (e^y - e^{-y})/2$

(11) $(z_1)^{(z_2)} = e^{(z_2)*\ln(z_1)}$

Routine Listing For: CA

```
01♦LBL "CA"
02 GTO IND 06
03♦LBL A
04♦LBL 01
05 XEQ 16
06♦LBL 00
07 XEQ 13
08 ST+ Z
09 X<> T
10 +
11 X<>Y
12 GTO 10
13♦LBL B
14♦LBL 02
15 XEQ 16
16 X<>Y
17 CHS
18 X<>Y
19 CHS
20 GTO 00
21♦LBL C
22♦LBL 03
23 XEQ 16
24♦LBL 17
25 XEQ 13
26 STO [
27 X<> T
28 ST* [
29 X<>Y
30 *
31 X<>Y
32 LASTX
33 X<>Y
34 ST* T
35 *
36 RCL [
37 +
38 R↑
39 RCL Z
40 -
41 GTO 10
42♦LBL D
43♦LBL 04
44 XEQ 16
45 STO Z
46 X↑2
47 RCL Y
48 X↑2
49 +
50 ST/ Z
51 /
52 CHS
53 X<>Y
54 GTO 17
55♦LBL E
56♦LBL 05
57 XEQ 16
58 R-P
59 LN
60 GTO 10
61♦LBL 06
62 9.009
63 STO 09
64 RTN
65♦LBL 07
66 2
67 RCL Z
68 ST+ X
69 E↑X-1
70 +
71 LASTX
72 R↑
73 CHS
74 E↑X
75 ST* Z
76 *
77 2
78 ST/ Z
79 /
80 RTN
81♦LBL H
82♦LBL 08
83 XEQ 16
84 XEQ 07
85 R↑
86 COS
87 *
88 X<>Y
89 R↑
90 SIN
91 *
92 GTO 10
93♦LBL I
94♦LBL 09
95 XEQ 16
96 XEQ 07
97 R↑
98 SIN
99 *
100 CHS
101 X<>Y
102 R↑
103 COS
104 *
105 GTO 10
106♦LBL a
107♦LBL 11
108 XEQ 13
109 XEQ 13
110 R↑
111 R↑
112 XEQ 10
113 R↑
114 R↑
115 GTO 10
116♦LBL b
117♦LBL 12
118 XEQ 11
119 XEQ 05
120 XEQ 03
121♦LBL e
122♦LBL 15
123 XEQ 16
124 E↑X
125 P-R
126 GTO 10
127♦LBL 16
128 SF 10
129♦LBL c
130♦LBL 13
131 RCL IND 09
132 FS? 10
133 STO 08
134 DSE 09
135 RCL IND 09
136 FS?C 10
137 STO 07
138 X<>Y
139 DSE 09
140 RTN
141 ISG 09
142 ""
143 ISG 09
144 ""
145 RTN
146♦LBL d
```

Listing continued on page 80.

Listing continued from page 79.

Routine Listing For: CA	
147♦LBL 14 148 RCL 07 149 RCL 08 150♦LBL J 151♦LBL 10 152 X<>Y 153 ISG 09 154 "" 155 STO IND 09 156 X<>Y 157 ISG 09 158 "" 159 STO IND 09 160 END	

LINE BY LINE ANALYSIS OF CA

Line 02 provides access to all numeric labels within CA.

Lines 03-12 perform addition of the bottom two numbers on the complex stack.

Lines 13-20 are used to change the sign on the complex number to be subtracted. The subtraction routine then jumps into the middle of the addition routine.

Lines 21-41 perform multiplication of the bottom two numbers on the complex stack.

Lines 42-54 calculate the conjugate of the complex number on the bottom of the complex stack and then divide it by its length. The division routine then jumps into the middle of the multiplication routine.

Lines 55-60 calculate ln(z), formula (5).

Lines 61-64 initialize the stack by storing 9.009 in R09 = bottom of stack pointer.

Lines 65-80 are an internal subroutine which calculate cosh(y) and sinh(y), formulas (9) and (10). This subroutine and these values are used in the calculation of both sin(z) and cos(z).

Lines 81-92 calculate sin(z) where z is the bottom number on the complex stack.

Lines 93-105 calculate cos(z) where z is the bottom number on the complex stack.

Lines 106-115 perform an exchange between the two numbers on the bottom of the complex stack. This is the complex X<>Y subroutine.

Lines 116-126 calculate $(z_1)^{(z_2)}$, formula (11).

Lines 127-145 are essentially the Pop Stack routine where the complex number may be saved in LAST Z and the top stack element may be replicated.

Lines 146-149 recall the complex number from LAST Z.

Lines 150-160 are the Push subroutine.

NUMERIC LABELS/FUNCTIONS IN THE CA PROGRAM

The following list gives a correspondence between numeric labels and subroutines to be called as part of CA programs. To call a subroutine function from one of your own programs, first store the number corresponding to the desired function in data register R06. Then use the instruction XEQ "CA" as part of your program. Approximate execution times for these subroutines are given in seconds.

Numeric Label Number in R06	Keyboard Label	Subroutine Function
01	A	Addition (2.4 sec.)
02	B	Subtraction (2.5 sec.)
03	C	Multiplication (2.6 sec.)
04	D	Division (3.0 sec.)
05	E	ln(z) (2.5 sec.)
06	(XEQ 06)	Initialize Stack (<1 sec.)
07	none	cosh(y), sinh(y) (1.6 sec.)
08	H	sin(z) (3.7 sec.)
09	I	cos(z) (3.7 sec.)
10	J	Push onto complex stack (1.2 sec.)
11	a	Complex X<>Y (2.3 sec.)
12	b	Complex Y to the X power (7.2 sec.)
13	c	Pop without saving in LAST Z (<1 sec.)
14	d	Recall LAST Z (<1 sec.)
15	e	e^z (2.4 sec.)
16	none	Pop stack and save in LAST Z (<1 sec.)

Note that labels 07 and 16 are not represented by functions on the keyboard. The following shows the XYZT stack input/output for LBL 07.

INPUT:		OUTPUT:	
	T: T		T: X
	Z: Z		Z: X
	Y: Y		Y: cosh(Y)
	X: X		X: sinh(Y)
	L: L		L: 2

LBL 16 is an alternate version of LBL 13. Sometimes when popping the stack it is desirable to first save the complex number in LAST Z and at other times this may not be necessary. Labels 13 and 16 provide the user with a choice when to save or not save LAST Z.

REFERENCES FOR CA

1. R.V. Churchill, "Complex Variables and Applications," McGraw-Hill Book Co., 1960

2. Lars V. Ahlfors, "Complex Analysis," McGraw-Hill Book Company, 1966

3. Peter Ladrach (5060), PPC Calculator Journal, V7N4P10

4. Peter Van Den Hammer (3533), PPC Calculator Journal, Complex Stack Comparisons, V7N2P52

5. Bruce Murdock (2916), PPC Calculator Journal, V7N1P16

CONTRIBUTORS HISTORY FOR CA

The CA program and documentation were written by John Kennedy (918). Carl Rosenfeld (5591) suggested an improvement in the accuracy of the calculations for the hyperbolic functions. Another member (unknown?)

suggested an improvement over the telephone which made possible the complex stack replication.

FINAL REMARKS FOR CA

Many members have discussed the desirability of a 5 high stack or even higher and the CA program has implemented an infinite stack (also useful for extended precision) which should provide a capability whose desirability will be tested through time and use.

Although the complex arithmetic program is specifically designed to do calculations with complex numbers, its main feature is the use of an extended stack with pairs of numbers. Since the complex stack handles input/output of pairs of numbers, a natural extension of this use of the stack would be to input and output numbers in a double precision arithmetic program. Other users may wish to add other complex arithmetic functions that had to be left out of the PPC ROM because of space limitations.

FURTHER ASSISTANCE ON CA

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

NOTES

TECHNICAL DETAILS

XROM: 20, 23	CA	SIZE: 018 (variable)

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: scratch (* /)
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
- R00: not used
- R06: function call #
- R07: LAST Z (IM part)
- R08: LAST Z (RE part)
- R09: stack pointer
- R10: start of stack
- R11: complex stack
- R12:

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: controls saving LAST Z
- 25: not used

Display Mode: not used

Angular Mode: RADIANS mode

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine:
A, B, C, D, E, a, b, c, d, e, H, I, J, 00, 01, 02,03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 14, 15, 16, 17

Execution Time: see NUMERIC LABELS section in CA documentation

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: CA

Bytes In RAM: 262

Registers To Copy: 38

Other Comments:
See COMPLETE INSTRUCTIONS for exact SIZE requirements.

CB - COUNT BYTES

If you have a printer, executing CAT 1 in Trace mode gives a full listing of RAM programs with byte counts for each.

CB can provide you with this same information and more, with or without a printer. CB can be used to count bytes between any two program lines, not just between ENDs.

Example 1: Assign RCL b to any key (see MK Example 1).

GTO.. and key in the following program:

```
01 LBL "BUG"
02 "HP-41C"
03 CF 21
04 AVIEW
05 SF 25
06 CLX
07 1/x
08 LBL 00
09 SIN
10 GTO 00
```

Switch to RUN mode, GTO .001 and RCLb. GTO.. and RCL b again, then XEQ CB . The result is 28, indicating that the program "BUG" is 28 bytes including an END.

Example 2: The above example can be continued to show how to get a byte count excluding the END. GTO "BUG", RCL b, BST, RCLb, and XEQ CB . The result is a count of 25 bytes without the END.

Now that you've finished the examples, try executing "BUG". Unless HP fixed the bug, you'll see a scrolling "HP-41C". Try adding 10 more LBL 00's between lines 08 and 09 and compare the effect. Then try using six LBL 00's. See IF Example 6 for an explanation of this behavior.

COMPLETE INSTRUCTIONS FOR CB

CB will find the distance in bytes between two RAM program pointers. It will work even if there are subroutine return addresses included in the pointer. To use CB go to a program line (usually line 00 or 01), RCLb in RUN mode, go to another program line (usually END or line 01 of the next program), RCL b in RUN mode, and XEQ CB . The result will be the number of bytes from the first program line to the second program line. The bytes in the first program line are included in the count, while the bytes in the second program line are not. If the second program line was above the first, the result is negative.

CB leaves the result in X, the decimal equivalent of the first pointer in Y, Z, and T, and the decimal equivalent of the second pointer in L. Alpha is cleared. CB will not object to invalid inputs, but the results will not be meaningful.

MORE EXAMPLES OF CB

Example 3: Determine whether a two-byte GTO and a one byte label can be used without losing the advantages of a compiled branch distance (see Appendix G of the *OWNER'S HANDBOOK AND PROGRAMMING GUIDE*). First construct the GTO and LBL in their desired positions as GTO nn and LBL nn, where nn is between 00 and 14 (inclusive). Then PACK and count bytes as follows.

Go to the line following the GTO instruction (if it's .END. put in a dummy instruction and PACK) and RCL b in RUN mode. Then go to the LBL instruction (you can use BST, SST) and RCL b again. XEQ CB to see the jump distance in bytes. If this jump distance is between -111 and +111 bytes (inclusive), then the two byte GTO is sufficient. Otherwise you need a three-byte GTO.

An alternative procedure is to RCLb at the GTO instruction, SST to get to the LBL, RCLb, and XEQ CB. The result should be between -109 and +113, inclusive.

If you need a three-byte GTO you can construct a synthetic one using LB inputs 208, 0, and nn, where nn is between 00 and 14. This allows you to use the one-byte LBL nn, saving one byte over the standard instructions GTOxx, LBLxx, with 15<xx<99. Once created, a synthetic three-byte GTO will never change to a two-byte GTO and it will always compile the branch properly. It can only be distinguished from a two-byte GTO by byte jumping or byte counting.

APPLICATION PROGRAM 1 FOR CB

The ROM program pointer format differs from the RAM format as explained in the Ab write-up. Therefore, CB cannot be used to count bytes in a ROM program, at least until after it has been downloaded. The program "CBR" (Count Bytes in ROM), listed below, use PD Application Program 1 "RPD" (ROM pointer to Decimal) to implement a ROM byte counting capability precisely analogous to the RAM byte counting capability of CB.

```
01 LBL "CBR"
02 XEQ "RPD"
03 X<>Y
04 XEQ "RPD"
05 -
06 RTN
```

For example, if you RCLb at LBL CB in ROM and RCL b again at LBL RT , XEQ "CBR" yields a count of 14 bytes in ROM. This agrees with the count CB would give between these labels after downloading.

Routine Listing For:	CB
33◆LBL "CB" 34 X<>Y 35 XROM "PD" 36 X<>Y	37 XROM "PD" 38 - 39 RTN

LINE BY LINE ANALYSIS OF CB

Lines 35 and 37 convert the two program pointer to decimal byte counts from the bottom of RAM (see PD), then line 38 computes the difference.

TECHNICAL DETAILS

XROM: 10,50	**CB**	SIZE: 000

Stack Usage: 0 T: decimal #1 1 Z: decimal #1 2 Y: decimal #1 3 X: result 4 L: decimal #2	Flag Usage: NONE USED 04: 05: 06: 07: 08: 09: 10: 25:
Alpha Register Usage: 5 M: 6 N: ALL CLEARED 7 O: 8 P:	
Other Status Registers: 9 Q: 10 ⊢: 11 a: NONE USED 12 b: 13 c: 14 d: 15 e:	Display Mode: UNCHANGED Angular Mode: UNCHANGED Unused Subroutine Levels: 4
ΣREG: UNCHANGED Data Registers: NONE USED R00: R06: R07: R08: R09: R10: R11: R12:	Global Labels Called: Direct: PD Secondary: 2D, QR Local Labels In This Routine: NONE
Execution Time: 3.6 seconds.	
Peripherals Required: NONE	
Interruptible? YES Execute Anytime? NO Program File: IF Bytes In RAM: 14 Registers To Copy: 60	Other Comments:

NOTES

CONTRIBUTORS HISTORY FOR CB

The idea of a program to count RAM bytes originated with Charles Close (3878) in *PPC CALCULATOR JOURNAL*, V7N3P29d. Early versions were written by Bill Wickes (3735) in V7N3P7, Valentin Albillo (4747) in V7N5P17a, and Tom Cadwallader (3502) in V7N9P28. The ROM version of CB was written by Roger Hill (4940) as a simple use of PD, another of his programs.

FURTHER ASSISTANCE ON CB

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

CD - CHARACTER TO DECIMAL

CD is a character decoding subroutine that will handle up to 15 characters one by one. Each execution of **CD** decodes the rightmost character to an equivalent decimal number (from the byte table) between 0 and 255. This provides a capability similar to **NH**, but the decimal output makes **CD** more useful as a subroutine. In fact, the first version of **CD** was used in the first version of **NH**.

Example 1: Decode an alpha string into its decimal equivalents. Key "HP-41C" into alpha, clear flag 10, and XEQ **CD**. The result in X, 67, is the decimal equivalent of the character C. Remaining in alpha is "HP-41". XEQ **CD** five more times to decode the remaining characters. The results are 49, 52, 45, 80, and 72, corresponding to the characters 1, 4, -, P, and H.

COMPLETE INSTRUCTIONS FOR CD

With the desired string of up to 15 characters (14 with flag 10 set) in alpha, XEQ **CD**. This places the decimal equivalent (0 to 255) of the rightmost character in X. The previous contents of X, Y, and Z are raised to Y, Z, and T, respectively. If flag 10 is clear the rightmost character is removed from the string; it remains in place if flag 10 is set. The contents of L are replaced by 10.

DC is essentially the inverse of **CD**. This pair of routines can be very powerful when used together, as they are in Application Program 1.

MORE EXAMPLES OF CD

Example 2: Decode a selected character. Key ABCDEFGHIJKL into alpha. Select the 8th character (from the right) with 8 XEQ **NC**. Then XEQ **CD** to get 69, the decimal equivalent of "E".

Example 3: A PPC member is confused by the **LG** program and wants to know what the byte codes are that comprise lines 2 and 3. He decides to use the **CD** routine to convert the two synthetic text lines to HEX table values. The first step in this "decoding" process is to assign **CD** to a convenient key. The second step is to go to line two of **LG** and SST in RUN mode. This places the 14 character text line into the Alpha register. The last step is to repeatedly press the **CD** assigned key and write down the results right to left. Here is what he wrote down:

(254) 17, 194, 228, 124, 60, 122, 241, 17, 102, 62, 30, 61, 120, 249.

The **CD** routine provides the decimal number of the alpha character right to left, so it is most convenient to write it only once and do it the same way. The text character, TEXT 14 (254), is not an alpha character, but is required for the 41 to know that the following 14 bytes should be considered as alpha characters. Hence, the TEXT 14 byte is part of the program line, but not part of the resulting text line.

The alpha register is now clear, and line 3 is SST'd and decoded in the same way to produce: (248) (127) 17, 158, 29, 155, 191, 78, 135.

This line is a text append line and the lazy "T" append symbol must be added to tell the 41 that the bytes following the append are to be appended to the alpha register. The append "character" appears in the display. If the append "character" follows a TEXT 1 thru TEXT 15 instruction, it becomes an append instruction. The text instruction must include the append, and TEXT 8, 248, preceeds the alpha text string.

The numbers from the second half of the HEX table bother the member, so he decides to verify them by creating the textline himself using **LB**. First he executes COPY **LG** and isolates only the **LG** routine in RAM. He deletes lines 1, 2, & 3, and keys a new label "LBL LGG" using the three letter ROM related convention. Next he keys the **LB** required LBL ++, +, +, ...+, XEQ **LB** with 34 or more +'s. In RUN mode: R/S and the numbers 254, 17, 194, ...248, 127,... 135 are keyed in alternately with R/S. The extra +'s etc. are deleted and the program is run. Sure enough, the LOGO produced is the same as **LG**.

APPLICATION PROGRAM 1 FOR CD

The program ATR listed below will translate (encode or decode) an alpha string of 1 to 6 characters to another alpha string of the same length, calling a user-supplied program CIPHER to perform the translation in decimal.

```
01 LBL "ATR"
02 ⊦*
03 6
04 CF 10
05 LBL 00
06 ⊦*
07 X<>N
08 X<>M
09 XROM CD
10 X≠0?
11 XEQ "CIPHER"
12 XROM DC
13 X<>M
14 X<>N
15 DSE X
16 GTO 00
17 X<>N
18 STO M
19 RDN
20 RTN
```

The CIPHER program can be quite simple, for example,

```
LBL "CIPHER"
128
FS? 09
CHS
+
RTN
```

This example encodes (Flag 09 clear) by moving all characters to the second half of the hex table, and reverses the process (Flag 09 set) to decode. Unfortunately this code can be easily broken by the printer function PRA.

A fancier code using rotation of the alphabet and numbers is also relatively easy to implement:

```
01 LBL "CIPHER"
02 65
03 X>Y?
04 GTO 14
05 -
06 RCL 01
07 FS? 09
08 CHS
09 +
10 26
11 MOD
12 65
13 +
14 RTN
15 LBL 14
16 RDN
17 8
18 -
19 RCL 02
20 FS? 09
21 CHS
22 +
23 10
24 MOD
25 48
26 +
27 RTN
```

The CIPHER program accepts the decimal equivalents of its permissible character bytes and converts these to the decimal equivalents of encoded characters (or the reverse, for decoding). For an example of the use of ATR, key up ATR and the longer version of CIPHER. Then store 11 in register 01 and 3 in register 02. Key WB6QRM into alpha, clear flag 09 and XEQ "ATR". The result in alpha, HM9BCX, is the encoding resulting when the alphabet is shifted 11 characters and the numbers by three. Set flag 09 and XEQ "ATR" to decode. This simple rotation code is easily broken, but perhaps you can devise more elaborate ones.

APPLICATION PROGRAM 2 FOR CD

"Little CD " by Roger Hill (4940) works like CD , but it's faster. It is limited to 7 characters in alpha and preserves only two stack registers.

```
LBL "cd"
hex F7 7F 00 07 06 00 00 00
R↑
CLX
X<>N
hex F4 7F 00 09 99
X<>M
RCL N
INT
HMS
*
.
X<>N
+
E1
*
RTN
```

Line 02 is hex F7 7F 00 07 06 00 00 00
Line 04 is hex F4 7F 00 09 99

Routine Listing For: CD

```
178♦LBL "CD"      187 CLX         197 ST* ]
179 "├♦↓****"     188 X<> \       198 RDN
180 RCL [         189 "├***"      199  E1
181 FS? 10        190 X<> [       200 ST* L
182 "├*"          191 X<> L       201 X<> L
183 STO [         192 X<> \       202 ST+ ]
184 CLX           193 INT         203 CLX
185 X<> ]         194 ST+ ]       204 X<> ]
186 SIGN          195 RDN         205 END
                  196 6
```

LINE BY LINE ANALYSIS OF CD

Line 179 attaches the bytes 00 07 to the right of alpha, and adds 4 more dummy bytes to push what was the rightmost character into the leftmost byte of M. Lines 180-183 permit the former rightmost character to be preserved if flag 10 was set. Lines 184-188 transfer the 0 register to L and the N register to X. Line 189 pushes from M into N a 3-byte string consisting of the byte to be decoded followed by 00 07.

The 07 byte acts as an exponent to put the byte to be decoded in the form a.b. Lines 190 through 192 transfer L to N, X to M, and N to X. Lines 193 through 197 isolate the first nybble a, normalize it, and place 6a in the 0 register.

Lines 198-202 add 10a + b, leaving 16a + b in the 0 register. This is the decimal equivalent of the byte ab_{16}. Lines 203-205 clear the 0 register and extract the result.

CONTRIBUTORS HISTORY FOR CD

CD began as UNBLD (see *PPC CALCULATOR JOURNAL*, V6N8P29), one of the original Black Box programs by Bill Wickes (3735). It was used as a subroutine for the original DECODE program. The ROM version of CD, written by Roger Hill (4940), is shorter, faster, and allows up to 15 characters in alpha. It uses the INT function to separate the two nybbles of a byte. This technique, used in many synthetic programs, was introduced by Bill Wickes in an improved version of UNBLD (see *PPC CALCULATOR JOURNAL*, V7N2P36).

FURTHER ASSISTANCE ON CD

Call Carter Buck (4783) at (415)653-6901

Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,35 — **CD** — SIZE: 000

Stack Usage:
- 0 T: Z
- 1 Z: Y
- 2 Y: X
- 3 X: decimal 0-255
- 4 L: 10

Flag Usage: ONLY FLAG 10 IS USED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10: TESTED ONLY
- 25:

Alpha Register Usage:
- 5 M: FL 10: SET-UNCHANGED. CLEAR-SHIFTED
- 6 N: RIGHT ONE PLACE.
- 7 O: CLEARED
- 8 P: CLEARED

Other Status Registers: NONE USED
- 9 Q:
- 10 ⊦:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R11:
- R12:

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: 1.3 seconds.

Peripherals Required: NONE

- Interruptible? YES
- Execute Anytime? YES
- Program File: VM
- Bytes In RAM: 63
- Registers To Copy: 60

Other Comments:

CJ - CALENDAR DATE TO JULIAN DAY NUMBER

This is a calendar routine which computes the Julian Day Number (JDN) of a given date. The valid range is from March 1 of the year 0 A.D. (=1 B.C.). Gregorian or Julian calendar dates may be input depending on a flag setting. The input is of the form with the year in Z, the month in Y and the day number in X. See also the routine JC. This routine is the inverse of JC.

Example 1: Compute the Julian Day Number of July 4, 1981.

Clear flag 10 for Gregorian Calendar dates. Key 1981 ENTER↑ 7 ENTER↑ 4. The stack input is of the form:

Z: year = 1981
Y: month = 7
X: day = 4

XEQ "CJ". JDN = 2,444,790.

Example 2: Compute the day of the week of July 4, 1981.

The day of the week may be found by computing:

day of week code = (JDN + 1) MOD 7.

If 2,444,790 remains in X from the previous example key 1 + 7 MOD and see the number 6 returned. Following the standard convention that 0=Sunday, 1=Monday, 2=Tuesday, 3=Wednesday, ect., we find that July 4, 1981 should be Saturday (=6).

BACKGROUND FOR CJ

The Julian Day Number (hereafter called JDN) is the number of whole days that have elapsed since a certain reference time in the past. The JDN is widely used in astronomy and elsewhere in calculations involving the counting of days.

The reference time from which all JDN's are measured has been chosen by astronomers to be January 1, 4713 B.C. (Julian Calendar) at noon. Thus from noon January 1 to noon January 2, 4713 B.C., the JDN is 0; from noon January 2 to noon January 3 the JDN is 1 and so on. As an example of a JDN in modern times, from noon January 30 to noon January 31, 1982 (present Gregorian calendar) the JDN is 2,445,000.

The reason for starting at noon (rather than midnight, as in ordinary calendar dates) is so that the JDN will not change in the middle of a night, making it convenient for use in astronomical observations. This can cause some confusion, however, when it comes to identifying JDN's with calendar dates unless one defines his/her terms carefully. Strictly speaking, JDN 2,445,000 corresponds to calendar date 1982 Jan. 30.5 (i.e., halfway through Jan. 30), and 1982 Jan. 30 corresponds to JDN 2,444,999.5 (i.e., halfway through JDN 2,444,999; a JDN with fractional part included to indicate time of day is called the "Julian Date"). One can also ask whether the "noon" refers to local time, Greenwich meridian time, etc. However, in the CJ and JC routines and discussions thereof we shall (except as noted) only deal with integer-valued JDN's and calendar dates, with the understanding (whenever one needs to worry about such matters) that any JDN under discussion is the one beginning at noon on the corresponding calendar date. The user may then add his own fractional parts, time corrections, etc. Thus, 1982 Jan. 30 corresponds to JDN 2,445,000, and this will be the result obtained by executing routines CJ or JC.

It should be noted that the terms "Julian Day Number" and "Julian Date" are not always distinguished from each other in the literature, but we have adopted the above definitions for the sake of clarity and preciseness. The user who is confused by the preceding paragraph and only needs to calculate days between dates, etc., may simply forget the above discussion and just consider the JDN as a continuous numbering of days starting from some day in the remote past. One the other hand, it is necessary to be aware of the two types of calendars discussed in the next paragraph in order to be certain that the correct calendar is being used (as determined by the setting of Flag 10).

The terms "Julian Day Number" and "Julian Date" should not be confused with "Julian Calendar", the latter being the system of arranging days into months and years which came into being in early B.C. years (46 B.C. as decreed by Julius Caesar) and lasted until 1582 when the presently-used Gregorian Calendar was introduced. The JDN is independent of which calendar is in use, and in fact provides a convenient way of converting dates from one calendar system to another. Both the Julian and Gregorian Calendars have the familiar numbers of days in the months, with 365 days in a common year and 366 days in a leap year. The Julian Calendar had a leap year every four years (the years divisible by 4) making the average length of a year 365 and 1/4 days. The true length of a year, however, is about 365.2422 days (as determined by the earth's revolution around the sun compared to the earth's revolution about its own axis), and by the 1500's the error in the beginning of a calendar year compared to the beginning of a real year amounted to about ten days. The problem was corrected in 1582 by Pope Gregory XIII with the help of astronomer Luigi Lilio: ten days were removed from 1582 (Oct. 4 in the old system was followed by Oct. 15), and the leap year system was modified so that years divisible by 100 (which were leap years in the Julian Calendar) are no longer leap years unless they are also divisible by 400. Thus 1600 and 2000 are leap years, but 1700, 1800, and 1900 are not. This makes the average length of a calendar year 365.2425 days, closer to the true value. This system, the Gregorian Calendar, is the one used today, although not all countries adopted it when it was originally introduced.

The CJ and JC routines allow either calendar to be used for inputting and outputting calendar dates. If Flag 10 is clear, the Gregorian Calendar will be assumed, and if Flag 10 is set, the Julian Calendar will be assumed. It should be emphasized again that the Julian Day Number is independent of the calendar system being used; a given day in history has only one JDN but may have several calendar dates depending whether we are using the Julian, Gregorian, or some other calendar.

A few additional facts worth noting are (1) the Julian Calendar repeats (including the day of the week) every 28 years, while the Gregorian Calendar repeats every 400 years, and (2) the Julian and Gregorian Calendars coincide during the years 201-299 A.D., a fact made use of in the CJ and JC routines.

Note: The CJ and JC routines do not take into account a proposed modification to the Gregorian Calendar which would make all multiples of 4000 non-leap years--a further correction to the calendar year/day ratio to bring it closer to the actual ratio.

The British Empire, including the American Colonies, did not adopt the Gregorian Calendar until September 2, 1752, that date being followed by September 14, 1752. Many dates of American history have been converted from the Julian or Old Style (O.S.) Calendar to the Gregorian New Style (N.S.) Calendar. For example, George Washington was born February 11, 1732 (O.S.), but we celebrate it on February 22, 1732 (N.S.). Some other countries were even slower in adopting the Gregorian Calendar, the last being Turkey in 1927. The definitive list of dates countries adopted the Gregorian Calendar is found in reference 8. The Julian Calendar is still used for some purposes by the Eastern Orthodox Church. For dates in modern times, both the CJ and JC routines would normally be used with Flag 10 clear, indicating that the dates input/output are for the Gregorian Calendar.

COMPLETE INSTRUCTIONS FOR CJ

1) Clear flag 10 for Gregorian calendar dates. Set flag 10 for Julian calendar dates.

2) Input the date in the stack in the form:

Z: year
Y: month
X: day

The valid range is from March 1 year 0 A.D.

3) XEQ "CJ". The Julian Day Number (JDN) is returned in the X-register. LAST X will contain the constant 1721115. CJ does not preserve the stack. If "DATA ERROR" then a date before March 1 0 A.D. was input.

4) If the day of the week (DOW) is desired, add 1 to the JDN and then compute the remainder when this sum is divided by 7 (i.e., DOW = JDN + 1 MOD 7, where 0=Sunday, 1=Monday, 2=Tuesday, 3=Wednesday, 4=Thursday, 5=Friday, 6=Saturday).

MORE EXAMPLES OF CJ

Example 3: Compute the Julian Day Number of January 1, 2000.

Clear flag 10 and key 2000 ENTER↑ 1 ENTER↑ and XEQ "CJ". JDN = 2,451,545.

Example 4: Compute the Julian Day Number of October 4, 1582 under the Julian calendar.

Set flag 10 for the Julian calendar. Key 1582 ENTER↑ 10 ENTER↑ 4 and XEQ "CJ". JDN = 2,299,160.

Example 5: Compute the Julian Day Number of October 15, 1582 under the Gregorian calendar.

Clear flag 10. Key 1582 ENTER↑ 10 ENTER↑ 15 and XEQ "CJ". JDN = 2,299,161.

Example 6: Compute the JDN of Sept. 2, 1752 under the Julian calendar.

Set flag 10 for the Julian calendar. Key 1752 ENTER↑ 9 ENTER↑ 2 and XEQ "CJ". JDN = 2,361,221.

Example 7: Compute the JDN of September 14, 1752 under the Gregorian calendar.

Clear flag 10 for the Gregorian calendar. Key 1752 ENTER↑ 9 ENTER↑ 14 and XEQ "CJ". JDN = 2,361,222.

Example 8: Find the number of days between June 20, 1981 and September 15, 1981.

We apply CJ twice and take the difference between the Julian Day Numbers for these two dates. Clear flag 10. Key 1981 ENTER↑ 6 ENTER↑ 20 and XEQ "CJ". See 2,444,776. STO 01. Key 1981 ENTER↑ 9 ENTER↑ 15 and XEQ "CJ" and see 2,444,863. RCL 01 - . The number of days is 87.

Example 9: Find the day of the week of November 21, 1963.

Clear flag 10. Key 1963 ENTER↑ 11 ENTER↑ 21 and XEQ "CJ". See 2,438,355. Add 1 and then compute MOD 7. 1 + 7 MOD and see 4. The day was Thursday.

FURTHER DISCUSSION OF CJ

Range of Validity for CJ

Basically, routine CJ will give the correct Julian Day Number for any date in the Christian (A.D.) era -- actually, any date back to and including March of the year 0 A.D. (=1 B.C.). It is up to the user to decide whether the Gregorian or Julian Calendar is to be used and to set flag 10 accordingly, but if for example a date before 1582 is input with Flag 10 clear (indicating the Gregorian Calendar to be used), the routine will give the correct result for the JDN had the Gregorian Calendar been in effect. Similarly, the Julian Calendar will be correctly extrapolated after 1582 if the user wishes to so use it.

The CJ routine, in the interest of speed, does not check for illegal dates, except that it gives a DATA ERROR message for dates prior to March of 0 A.D. to remind the user that B.C. years are not accomodated here. (See also one of the application programs which extends the range of CJ into the B.C. years). The year, month, and day are all truncated to integers before processing, however, and any out-of-the-normal-range values for the month and day will be correctly extrapolated to the next or previous year or month (provided that the extrapolation does not go back earlier than March of 0 A.D.). Thus an input of:

Z: 1980 = year
Y: -2 = month
X: 37 = day number of month

will be interpreted as the date 1979 October 37 or 1979 November 6, and a month of 15 will be interpreted as March of the following year. The only such abnormal dates that may give erroneous results (but no error message) are those with a legal month and year but a negative day of the month which is so negative that the date is brought into the B.C. era; this would never occur with normal usage but the user should be aware of it if he/she wishes to try any tricks with large negative dates.

The maximum dates for which CJ is valid are limited only by the 10-digit precision of the calculator. Although the author has not made a complete, systematic test, the routine appears to give correct results for dates around 100,000 years in the future, but not for dates 1,000,000 years in the future; errors in the latter case are due to round-off errors occuring in the calculation of quantities such as 367*y' (see LINE BY LINE ANALYSIS OF CJ). At any rate, the program is certainly valid for dates far beyond those for which our present calendars are likely to be used. (See note about the year 4000 in the "background" section.)

APPLICATION PROGRAM 1 FOR CJ

The following sequence of steps will accept an input of the form YYYY.MMDD in the X-register, using the Gregorian Calendar to give the Julian Day Number in the X-register:

```
ENTER↑
FRC
E2
*
ENTER↑
FRC
E2
*
CF 10
XROM CJ          14 bytes
```

NOTE: The instruction E2, which behaves exactly the same as 1 E2 but is one byte shorter, can be synthesized using ROM program LB (decimal byte inputs 27, 18; if you synthesize both E2's at once insert a zero byte between them), by scanning the appropriate bar codes, or by other methods.

APPLICATION PROGRAM 2 FOR CJ

The following sequence will accept an input of the form MM.DDYYYY, using the Gregorian Calendar to give the Julian Day Number in the X-register:

```
ENTER↑
FRC
E2
*
ENTER↑
FRC
E4
*
X<>Z
X<>Y
CF 10            17 bytes
XROM CJ
```

APPLICATION PROGRAM 3 FOR CJ

The following sequence will do "date subtraction" as follows: Given Y = Final YYYY.MMDD and X = initial YYYY.MMDD, the result after execution will be the number of days from the initial to the final date in X, the initial JDN in Last X, and the Final JDN in R00 (R00 may be replaced by any other unused register, and Flag 09 may be replaced by any other unused flag.) Again the Gregorian Calendar is used.

```
CF 10
SF 09
LBL 00
STO 00
X<>Y
ENTER↑
FRC
E2
*
ENTER↑
FRC
E2
*
XROM CJ
RCL 00
X<>Y
FS?C 09          26 bytes
GTO 00
—
```

APPLICATION PROGRAM 4 FOR CJ

The following sequence will do "date subtraction" as in application program 3, but the input dates are of the form MM.DDYYYY. (This is simply application 3 modified in the same way that application 2 is obtained from application 1.)

```
CF 10
SF 09
LBL 00
STO 00
X<>Y
ENTER↑
FRC
E2
*
ENTER↑
FRC
E4
*
X<>Z
X<>Y
XROM CJ
RCL 00
X<>Y
FS?C 09
GTO 00
—
                 29 bytes
```

APPLICATION PROGRAM 5 FOR CJ

The following sequence will accept an input of the form DDMM.YYYY, using the Gregorian Calendar to give the Julian Day Number in the X-register:

```
FRC
LAST X
E4
ST * T
SQRT
ST / Z
MOD
X<>Y
CF 10            16 bytes
XROM CJ
```

Using similar methods one can accomodate other formats such as DD.MMYYYY, MMDD.YYYY, etc., and the "date subtraction" application routines 3 and 4 can be altered accordingly.

APPLICATION PROGRAM 6 FOR CJ

Given the date in the form YYYY.MMDD in the Y-register, and the time in the form HH.MMSS in the X-register, the following sequence of operations will produce the Julian Date in the X-register. (Any other unused register may be used instead of R00).

```
STO 00
X<>Y
ENTER↑
FRC
E2
*
ENTER↑
FRC
E2
*
XROM CJ
.5
-
RCL 00
HR
24
/
+                        23 bytes
```

(Note: The time is assumed to be according to a 24-hour clock; e.g. 13.0000 = 1:00 PM)

WARNING: Since the calculator can only hold 10 digits of precision, and Julian Day Numbers for modern times have 7 digits, the smallest time interval that can be distinguished in this procedure (encoding the JDN and time into a single number) is .001 day, or 86.4 seconds. Round-off errors may make this even larger. If greater accuracy is desired, one can subtract a constant from the JDN before incorporating the time, thus in effect measuring all times from a more recent reference date than Jan. 1, 4713 B.C. (Astronomers sometimes use a "Modified Julian Date" which is the Julian Date minus 2,400,000.5.)

APPLICATION PROGRAM 7 FOR CJ

Given a year in the X-register, the following sequence will view and/or print on one line all of the months (Jan = 1, etc.) which contain a Friday the 13th (Gregorian Calendar) during that year. The routine makes use of the fact that if the 13th is on Friday, then the 2nd must be on Monday, which means that the JDN for the 2nd must be divisible by 7 (see above instructions).

```
CF 10        2
STO 00       XROM CJ
1.012        7
STO 01       MOD
FIX 0        X=0?
CF 29        "⊢ "
CLA          X=0?
ARCL 00      ARCL 01
"⊢ : "       ISG 01
LBL 00       GTO 00
RCL 00       XROM VA
RCL 01       BEEP          42 bytes
```

As an example, entering 1981 and executing this sequence will result in the following display and/or printout:

1981: 2 3 11

indicating that the Friday-the-thriteenths for 1981 are in February, March, and November. This routine is certainly not the fastest for accomplishing the task (as it calculates each month from scratch), but it is undoubtedly one of the most straightforward and economical of bytes of RAM--assuming of course that the PPC ROM is in place.

APPLICATION PROGRAM 8 FOR CJ

Program CJA is used in exactly the same manner as CJ but with the following additional features. BC years are entered as negative numbers. For years between 3/1/9600 BC and 12/31/1 AD, CJA adds 9601 to the year and subtracts 3506400 from the corresponding Julian Day Number. For astronomical sequencing of BC years (2AD, 1AD, 0AD, 1BC, 2BC), set Flag 4. It will remain set after execution. Flag 10 will be automatically set by CJA for years prior to 1582, and will be cleared after execution. Flag 10 must still be manually set for Julian Calendar dates after 1581. To facilitate chaining of operations such as finding the number of days between two dates, program CJA insures that the original contents of the X-register prior to entering the date ends up in the Y-register. If DATA ERROR is displayed, the year was before 3/1/9600 BC.

Example using CJA: Find the number of days between May 14, 637 BC (astronomical sequencing) and January 1, 1950.

Do:	See:	Result:
SF 04		Set F4 for astronomical seq.
1950 ENTER↑ 1 ENTER↑ 1 XEQ "CJA"		
	2433283	Julian Day Number for January 1, 1950
673 CHS ENTER↑ 5 ENTER↑ 14 XEQ "CJA"		BC years are negative
	1475744	
-	957539	# days difference between dates input

APPLICATION PROGRAM FOR: CJ

01♦LBL "CJA"	18 RDN
02 CF 09	19 9601
03 R↑	20 ST+ Y
04 STO [	21 SF 09
05 X<> T	22♦LBL 01
06 1582	23 RDN
07 X>Y?	24 STO T
08 SF 10	25 RDN
09 SIGN	26 XROM "CJ"
10 X<Y?	27 RCL [
11 GTO 01	28 X<>Y
12 X>Y?	29 3506400
13 FS? 04	30 FC?C 09
14 FS? 54	31 CLX
15 ST- Y	32 -
16 X=Y?	33 CF 10
17 ST- Y	34 END

APPLICATION PROGRAM 9 FOR CJ

CJ may be used to determine the local mean sidereal time (LMST) for the years 1950 to 2010 with better than a one second accuracy. LMST is used for determining the position of the planets or the stars.

Once the Julian Day Number (JDN) is found for a particular date, the LMST (star time) can be quickly derived from the Coordinated Universal Time (UTC) and the longitude. The formula from the 1981 Astronomical Almanac is:

LMST = ((JDN - 2,444,605)*24 + UTC)*1.002737909 +
6.6383321 - Longitude

where the Longitude is in decimal hours.

Julian Day Number returned by CJ is assumed for Greenwich noon and this is accounted for in the program. The LMST is reduced to the range 0 to 24 hours with the MOD function, which also works on negative numbers. The factor 1.002737909 is the difference in rates between UTC and mean sidereal time.

The actual or apparent sidereal time only varies by ±1.2 seconds from the mean sidereal time due to nutation of the earth's axis over an 18 year period (W. M. Smart, Text-book on Spherical Astronomy, 5th Ed., Cambridge at the University Press, 1962).

Example: Calculate the LMST on July 4, 1976 at 16.35 hours for Montague, MA, where the longitude is 72.3210.

Do:	See:	Result:
XEQ "LONG"	"DDD.MMSS,WEST+"	Prompt to enter Longitude in degrees, minutes, seconds, with West longitudes being positive.
72.3210 R/S	4.8357	Longitude in decimal hours for Montague, MA.
XEQ "LMST"	"UT IN HH.MMSS"	Prompt to enter Universal Time
16.35 R/S	6.3605	LMST = 6H36M05S

APPLICATION PROGRAM FOR: CJ

```
01+LBL "LONG"          16 24
02 "DDD.MMSS, WEST+"   17 *
03 PROMPT              18 +
04 HR                  19 1.002737909
05 15                  20 *
06 /                   21 6.6383221
07 STO 01              22 +
08 RTN                 23 RCL 01
09+LBL "LMST"          24 -
10 "UT IN HH.MMSS"     25 24
11 PROMPT              26 MOD
12 HR                  27 STO 02
13 RCL 03              28 HMS
14 2444605             29 RTN
15 -
```

For other routines using CJ, see the examples of routines using JC.

FORMULAS USED IN CJ

There are many algorithms available for converting a year, month, and day into a day number; the one used here seems to involve a minimum of time- and byte-consuming digit-entry instructions and has a minor advantage of correctly interpreting "out-of-range" days and months (see "Range of Validity"), should the user have occasion to input such dates.

Suppose February always had 30 days; then each year would be 367 days long -- call this an "extended year" and let an extended year be considered as beginning with March and ending with February. If Y, M, and D are the real year, month, and day-of-month (as input by the user), then the quantity:

$$y = Y + (M-3)/12$$

has an integer part corresponding to the "extended year" and a fractional part denoting the month of the extended year. Now consider the quantity z = 367*y which we can write as: 367*INT(y) + 367*FRC(y). (Note: INT and FRC refer to integer and fractional parts, as in the 41C instruction set. Also, the variables y, z, etc., mentioned here have nothing to do with the HP-41C's stack unless otherwise mentioned.) The 367*INT(y) gives us the expected 367 days per extended year, while the 367*FRC(y) behaves according to Table 1.

Table 1. Behavior of 367*FRC(y)

Month	FRC(y)	367*FRC(y)	No. of days*
Mar	0	0	0
Apr	1/12	30 7/12	31
May	2/12	61 2/12	61
Jun	3/12	91 9/12	92
Jul	4/12	122 4/12	122
Aug	4/12	152 11/12	153
Sep	6/12	183 6/12	184
Oct	7/12	214 1/12	214
Nov	8/12	244 8/12	245
Dec	9/12	275 3/12	275
Jan	10/12	305 10/12	306
Feb	11/12	336 5/12	337

*In an extended year prior to the given month, assuming all months have the same length as in a real year (except for February).

An inspection of this table will reveal that if any number between 7/12 (inclusive) and 8/12 (exclusive) is added to each of the numbers in the third column, and the integer part taken, then the result will be the same as the fourth column which is based on the real lengths of months. (The lower limit 7/12 is imposed by February, and the upper limit 8/12 by July.) Hence the quantity INT(367*y + p) will give the number of days prior to the given month (starting with March of 0 A.D. and assuming all years are extended years), provided that 7/12 <= p < 8/12.

Real years, however, have 1 or 2 days less than our idealized "extended years". Subtracting 2*INT(y) from the expression described above will remove 2 days from the end of each extended year (i.e., 2 days from February--which is why we defined the extended years to end with that month) and adding INT(y/4) will bring back one day at the end of every 4-extended-year period putting the leap-year days in their proper places. For the Gregorian century corrections, subtracting INT(y/100) will remove the leap-year day from century years, and adding INT(y/400) will return it to years divisible by 400. Thus we obtain the number of days, starting with March of 0 A.D., prior to the given month in the Gregorian Calendar. The day before March 1 of 0 A.D. happens to have JDN 1,721,119, which we add to our expression, and finally the day-of-month D is added to bring us to the input date, resulting in the following formula:

(1) JDN = INT(367*y + p) - 2*INT(y) + INT(y/4)
- INT(y/100) + INT(y/400) + D + 1,721,119

where y = Y + (M-3)/12.

In the actual program it turned out to be worthwhile to make a few modifications in the above formula. The quantity Q - INT(y) + INT(y/4) can be replaced by INT(Q - .75*INT(y)) as long as Q is an integer and is large enough so that the INT function is not applied to any negative numbers.

Also, Q - INT(y/100) + INT(y/400) can be replaced by INT(Q - .75*INT(y/100)) with the same restrictions on Q.

Furthermore, instead of INT(367*y + p) we can use INT(367*y') - n where y' = Y + (M - q)/12 and n is any non-negative integer, provided that the number q is between 3 - (8+12n)/367 (exclusive) and 3 - (7+12n)/367 (inclusive). (This restriction on q follows directly from the restriction on p mentioned above.) Choosing n=4 leads to:

2.847411 < q <= 2.850136

allowing q=2.85 to be used, and also ensures that INT will not be applied to negative numbers in the final result below. We can replace INT(y) by INT(y'), and replace INT(y/100) by INT(y'/100), as long as q is greater than 2 (which it is according to our above choice). The modified formula is then:

(2) JDN = INT(INT(INT(367*y') - INT(y')
- .75*INT(y') + D) - .75*INT(y'/100))
+ 1,721,115

where y' = Y + (M - 2.85)/12

For the Julian Calendar the quantity INT(y'/100) is simply replaced by 2, this being possible because the two calendars coincide during most of the 200's (see the "background" section).

Routine Listing For: CJ

```
118♦LBL E
119♦LBL "CJ"
120 INT
121 X<>Y
122 INT
123 2.85
124 -
125 12
126 /
127 R↑
128 INT
129 +
130 X<0?
131 SQRT
132 ENTER↑
133 INT
134 ST- Z
135 X<>Y
136 367
137 *
138 INT
139 ST+ Z
140 SIGN
141 FS? 10
142 ISG X
143 %
144 INT
145 .75
146 ST* Z
147 *
148 RDN
149 -
150 INT
151 R↑
152 -
153 INT
154 1721115
155 +
156 RTN
```

LINE BY LINE ANALYSIS OF CJ

In the routine CJ we start with the year, month, and day in the Z-, Y-, and X-registers of the stack and take the integer parts of these quantities (lines 128, 122, and 120) before doing anything else with them in order to avoid any errors or misinterpretations with fractional parts; the results are the Y, M, and D fed into the above formula. The quantity y' is calculated in lines 123-129, and lines 130-131 produce a DATA ERROR message if y' is negative (indicating a month before March of 0 A.D.). After line 140 the stack contents are:

Z: D - INT(y') + INT(367*y')
Y: INT(y')
X: 1

Now if Flag 10 is clear (Gregorian Calendar), then line 142 is skipped and after line 144 the X-register contains INT(y'/100). On the other hand, if Flag 10 is set (Julian Calendar), then the 1 in the X-register is changed to 2 by line 142 and line 143 is skipped, thus leaving 2 in the X-register after line 144. In lines 145-153 the Y- and X-registers get multiplied by .75 and subtracted from the Z-register, the integer part being taken after each subtraction, and finally we add the constant 1721115 in lines 154-155 to obtain the correct Julian Day Number.

REFERENCES FOR CJ

1. "Gregorian Calendar" 65 NOTES, HP-25 Library V4N5P20,P26

2. Dan M. Fenstermacher, "Calendar Algorithms", PPC JOURNAL V5N1P16

3. "CALENDARS", User's Library Solutions, For The HP-67/97, The Hewlett-Packard Company

4. American Journal of Physics, Vol. 49 No. 7, July 1981, pp. 658-661, "The Origin of the Julian Period".

5. "Calendar", Encyclopedia Brittanica (Contains much detailed information on various calendar systems, the calculation of Easter, etc.)

6. Gordon Moyer, "The Origin of the Julian Day System", SKY AND TELESCOPE, V61 N4 P311 (April 1981) Also contains algorithms for converting to and from Julian Day Number. (Corrections in June 1981, p. 550, and in July 1981, letter to the editor, p. 16)

Other general sources on Julian Day Numbers and astronomical time systems:

7. THE ASTRONOMICAL EPHEMERIS (previously, THE AMERICAN EPHEMERIS AND NAUTICAL ALMANAC), U.S. Government Printing Office, Washington, D.C. (any year)

8. Explanatory Supplement to THE ASTRONOMICAL EPHEMERIS and THE AMERICAN EPHEMERIS AND NAUTICAL ALMANAC, 1961, p.414 (and other editions)

9. C.W. Allen, ASTROPHYSICAL QUANTITIES, Athlone Press, London, 1976 (Examples of JDN on p. 295) 3rd Edition

CONTRIBUTORS HISTORY FOR CJ

The CJ routine and documentation is by Roger Hill

(4940). Earlier contributions were based on work by Fred Wheeler (1150) and by Fernando Lopez-Lopez (2887) but Roger came up with shorter and more comprehensive routines. David Spear (5488) provided some additional information on calendars and is the author of the application program CJA. Read Predmore (5184) is the author of the application program for the LMST.

FINAL REMARKS FOR CJ

The algorithm developed here is optimized for the HP-41C, but can be applied to other machines.

FURTHER ASSISTANCE ON CJ

Roger Hill (4940) phone: (618) 656-8825
Fernando Lopez-Lopez (2887) phone: (714) 421-9791 after 9PM

NOTES

TECHNICAL DETAILS

XROM: 20, 21 | **CJ** | SIZE: 000 (minimum)

Stack Usage:	Flag Usage:
0 T: used	04: not used
1 Z: used	05: not used
2 Y: used	06: not used
3 X: used	07: not used
4 L: used	08: not used
Alpha Register Usage:	09: not used
5 M: not used	10: clear Gregorian set Julian
6 N: not used	
7 O: not used	
8 P: not used	25: not used
Other Status Registers:	**Display Mode:**
9 Q: not used	not used
10 ⊦: not used	
11 a: not used	**Angular Mode:**
12 b: not used	not used
13 c: not used	
14 d: not used	**Unused Subroutine Levels:**
15 e: not used	5

ΣREG: not used	Global Labels Called:
Data Registers:	Direct: none / Secondary: none
R00:	
no data registers are used	
R06:	
R07:	
R08:	
R09:	
R10:	Local Labels In This Routine:
R11:	E
R12:	

Execution Time: approximately 1.9 seconds

Peripherals Required: none

Interruptible? yes	Other Comments:
Execute Anytime? no	
Program File: BD	
Bytes In RAM: 66	
Registers To Copy: 53	

STACK AND ALPHA REGISTER ANALYSIS FOR **CD**

T	Z	Y	X	L	P	O	N	M	L-#	INSTRUCTION
t	z	y	x	1	1 2 3 4 A B C	D E F G H I J	K L M N O P Q	R S T U V W X	178	♦LBL "CD"
z	y	x	X-⊠****		C D E F G H I	J K L M N O P	Q R S T U V W	X - ⊠ * * * *	179	"├♦↓****"
									180	RCL [
									181	FS? 10
					D E F G H I J	K L M N O P Q	R S T U V W X	- ⊠ * * * * *	182	"├*"
								X - ⊠ * * * *	183	STO [
			0						184	CLX
			KLMNOPQ			Cleared			185	X<>]
			-	KLMNOPQ					186	SIGN
			0						187	CLX
			RSTUVWX				Cleared		188	X<> \
					G H I J		X - ⊠	* * * * * * *	189	"├***"
			*******					R S T U V W X	190	X<> [
			KLMNOPQ	*******					191	X<> L
			a.b				K L M N O P Q		192	X<> \
			a	a.b					193	INT
						a			194	ST+]
a	z	y	x						195	RDN
z	y	x	6						196	6
						6a			197	ST*]
6	z	y	x						198	RDN
z	y	x	10						199	E1
				10a+b					200	ST* L
			10a+b	10					201	X<> L
						16 a +b			202	ST+]
			0						203	CLX
			16a+b			Cleared			204	X<>]
z	y	x	16a+b	10	G H I J	Cleared	K L M N O P Q	R S T U V W X	205	END

CK - CLEAR KEY ASSIGNMENTS

CK will clear any existing user key assignments of functions or programs. It also clears all other registers between the status registers and the permanent .END. (noninclusive). It is a fast, convenient alternative to individually clearing each key.

Example 1: Occasionally, unwanted program bytes are stored inadvertently below the .END.. This will result in the "blocking off" of program registers that would otherwise be available for use. In the extreme case, bytes stored in the register just below the .END. will result in the display "00 REG 00" (after GTO..), despite the fact that many free registers are available below the one containing extraneous bytes. CK provides relief from this condition, by clearing all registers below the .END.

COMPLETE INSTRUCTIONS FOR CK

CK requires no input. It may be executed either manually or as a program routine. In either case, upon execution of the routine, the key assignment registers--that is, the registers from the bottom of user memory, $0C0_{16}$, up to, but not including the register containing the permanent .END., and the key assignment flag registers ⊦ and e, will be cleared. To regain use of assigned global labels, read in a program card or status card. A program card containing only deleted lines is okay, and USER mode need not be set.

CK saves X, but loses Y, Z, T, and L. ALPHA is cleared.

Routine Listing For: CK

```
40♦LBL "CK"     46 177          52 STO ⊦        57 GTO 06
41 XROM "E?"    47 +            53 STO e        58 X<>Y
42 17           48 DSE X                        59 X<> c
43 -            49 RTN          54♦LBL 06       60 R↑
44 E3           50 XROM "OM"    55 STO IND Z    61 RTN
45 /            51 .            56 ISG Z
```

LINE BY LINE ANALYSIS OF CK

Lines 040 - 048 place in the X-register the decimal number 176.pqr, where pqr is the decimal value, less 17, of the absolute address of the register containing the permanent .END.. If user memory were filled to capacity, that register would be $0C0_{16} = 192_{10}$, and pqr would be 175. In this (and only this) case, line 049 will not be skipped, and the routine will be terminated at that point. In all other cases, it is possible that key assignment (or other) information is contained in registers below that containing the .END., and the routine will continue.

Line 50 makes use of the PPC ROM routine OM (Open Memory) to lower the "curtain" defining the beginning (R00) of the user data register block to the fictitious address 010_{16}. All of the user memory registers can then temporarily be accessed indirectly as data registers, beginning with $0C0_{16}$ = R176.

Lines 51-53 clear the key assignment flag (status) registers ⊦ and e.
Lines 54-57 consist of a loop that successively clears the key assignment registers beginning with R176 up to and including R_{pqr}, the register just below that containing the .END.

Lines 58-61 complete the routine by restoring to register c the code that was contained therein prior to execution of OM, thus restoring the "curtain" to its original location.

TECHNICAL DETAILS

XROM: 10,06 | **CK** | SIZE: 000

Stack Usage:
- 0 T: (E?-16).(E?-17)
- 1 Z: 0
- 2 Y: temporary c
- 3 X: X
- 4 L: 177

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

ALL CLEARED

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: CLEARED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED BUT RESTORED
- 14 d: USED BUT RESTORED
- 15 e: CLEARED

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: SEVERAL USED BUT ALL RESTORED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
E?	2D
OM	PART OF GE

Local Labels In This Routine: 06

Execution Time: 3 + .13 (E? + A?) seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? YES
Program File: LF
Bytes In RAM: 40
Registers To Copy: 59

Other Comments:

REFERENCES FOR CK

Earlier versions of CK and similar routines can be found in *PPC CALCULATOR JOURNAL*, V7N6P10b and V7N7P15b.

CONTRIBUTORS HISTORY FOR CK

CK was written by Keith Jarett (4360). Roger Hill (4940) modified it to save some bytes. Valentin Albillo (4747) independently conceived and wrote an early version called " CA ".

FURTHER ASSISTANCE ON CK

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825

NOTES

CM - COMBINATIONS

This routine will compute the number of combinations of n objects taken k at a time. This number may be denoted by C(n,k) and may be described as the number of all subsets (order doesn't count) of size k selected from a set of n objects. More formally,

C(n,k) = n!/[k!(n-k)!]

For CM the values n and k must satisfy the restriction 1<=k<=n. To minimize overflow errors and improve execution time this routine exploits the property that C(n,k) = C(n,n-k).

Example 1: Compute C(20,5)

Key 20 ENTER↑ 5 and XEQ "CM". C(20,5) = 15,504

COMPLETE INSTRUCTIONS FOR CM

1) To compute C(n,k) key n ENTER↑ k where 1<=k<=n.

2) XEQ "CM". The value C(n,k) will be returned in X. The value returned will not be exact if displayed in scientific notation. In this case however, the result displayed will be an accurate approximation.

The stack input/output for CM is as follows:

Input:		Output:	
T:	T	T:	n or n-1
Z:	Z	Z:	n
Y:	n	Y:	n
X:	k	X:	C(n,k)
L:	L	L:	0

MORE EXAMPLES OF CM

Example 2: How many 5-card poker hands can be dealt from a deck of 52 cards?

Key 52 ENTER↑ 5 and XEQ "CM". C(52,5) = 2,598,960.

Example 3: Assume that a basketball team will start 5 players who can play any position. How many starting line ups may be chosen from a group of 12 players?

Compute C(12,5). Key 12 ENTER↑ 5 and XEQ "CM". C(12,5) = 792

Example 4: A piano has 88 keys. How many chords of 4 sounds are possible if a chord is obtained by pressing 4 keys simultaneously?

We need to compute C(88,4). Key 88 ENTER↑ 4 and XEQ "CM". C(88,4) = 2,331,890.

FORMULAS USED IN CM

The only formula used in the CM routine is:

C(n,k) = n!/[k!(n-k)!]

However, it can be shown that C(n,k) = C(n,n-k) and it is more efficient to use the smaller of k or n-k as the second argument. The CM routine chooses the optimal value which reduces both execution time and round-off error when there is a possibility of overflow. A more meaningful form of the formula for C(n,k) that more closely resembles the program lines is:

$$C(n,k) = \frac{n*(n-1)*(n-2)* \ldots *(n-(k-1))}{k*(k-1)*(k-2) \ldots 3*2*1}$$

Routine Listing For: CM

```
97♦LBL D
98♦LBL "CM"
99 RCL Y
100 RCL Y
101 X≠Y?
102 -
103 X>Y?
104 X<>Y
105 ST+ T
106 SIGN
107 X<>Y
108♦LBL 08
109 X<> T
110 LASTX
111 ST- Y
112 /
113 ST* Y
114 DSE L
115 GTO 08
116 RDN
117 RTN
```

LINE BY LINE ANALYSIS OF CM

Lines 97-107 initialize the program. Lines 103 & 104 choose the optimal value of the second argument. Lines 106 and 107 store the initial value 1 in the Y-register which will hold the above partial products that result in the final answer.

Lines 108-115 are the main loop in the routine. At line 108 the partial answer is assumed to be in Y and a scratch value remains in X. Except for the first pass through the loop the T register holds n. LAST X holds a counter which may be called j. Initially j=k and j is decremented by one each time through the loop. The next partial product is formed by multiplying by the factor (n-j)/j. Line 114 tests to terminate the loop.

Lines 116-117 end the routine by returning the final answer in the X-register. The original n is returned in Y and Z. T contains a scratch value when the routine ends.

CONTRIBUTORS HISTORY FOR CM

The CM routine and documentation were written by John Kennedy (918).

FINAL REMARKS FOR CM

A future CM routine might extend the range of input arguments to include 0. This feature is not present in CM due to limited space in the ROM.

FURTHER ASSISTANCE ON CM

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

NOTES

TECHNICAL DETAILS		
XROM: 20, 20	CM	SIZE: none required

Stack Usage: 0 T: used 1 Z: used 2 Y: used 3 X: used 4 L: used	Flag Usage: 04: not used 05: not used 06: not used 07: not used 08: not used 09: not used 10: not used 25: not used
Alpha Register Usage: 5 M: not used 6 N: not used 7 O: not used 8 P: not used	
Other Status Registers: 9 Q: not used 10 ⊦: not used 11 a: not used 12 b: not used 13 c: not used 14 d: not used 15 e: not used	Display Mode: not used Angular Mode: not used Unused Subroutine Levels: 5
ΣREG: not used Data Registers: R00: R06: no data registers are used R07: R08: R09: R10: R11: R12:	Global Labels Called: Direct: none Secondary: none Local Labels In This Routine: D, 08

Execution Time: data dependent with a typical range less than 1 second to over 5 seconds

Peripherals Required: none

Interruptible? yes Execute Anytime? no Program File: BD Bytes In RAM: 37 Registers To Copy: 53	Other Comments: OUT OF RANGE message indicates too large inputs. SCI display mode indicates overflow but may still give valid approximation

CP - COLUMN PRINT FORMATTING

CP

This routine aligns numeric data into columns for printed output of tables or lists. A single skip index for each numeric column keeps decimal points in constant position. While CP only adds a single numeric column to the printer buffer, it may be called repeatedly to build multiple columns across the 24 character printed line. In addition, columns of ALPHA information may be accumulated by conventional ACA techniques to create virtually any combination of multiple numeric and ALPHA columns in printer output. Since the routine adds information to the print buffer without printing, CP may be called at any time during the creation of printed output.

Example 1. Print the numeric data from table 1 on the 82143A printer, aligning the columns with the CP routine.

```
  3.21   2 1,304.5 3.-06
 43.26   8 6,814.3 1.+30
  0.58  10 1,313.1 6.-09
618.18   1 4,441.6 3.-12
```

	Col. 1	Col. 2	Col. 3	Col. 4
Display mode	FIX 2	FIX 0	FIX 1	ENG 0
Commas?	NO	NO	YES	NO
Max. no. digits to the left of the dec. point:	3	2	4	1

Table 1. Four columns of numbers to be printed using the CP routine in example 1.

To print the data in table 1, we must first consult the detailed instructions below.

COMPLETE INSTRUCTIONS FOR CP

We must first plan the structure and size of each numeric column, as done below the columns in table 1. Next, we obtain the skip index for each numeric column. This is the number which CP uses to maintain the decimal points in constant position from line to line. Table 2 shows how to obtain the skip index for columns printed in FIX display format:

Max. No. of digits to the left of the decimal point in column. (NOTE: The HP41C won't allow zero digits!)	No Commas	Commas
1	0	0
2	1	1
3	2	2
4	3	4
5	4	5
6	5	6
7	6	8
8	7	9
9	8	10
10	9	12

Skip index (Add 1 for each extra space to the left of the sign position)

Table 2. Skip index values for CP in FIX display mode. From the presence or absence of commas and the maximum number of anticipated digits to the left of the decimal point in a column, a skip index may be chosen which will assure column alignment. Remember to add 1 to the index for each additional empty space to the left of the sign position in the numeric column. A sign position will always be created by CP for the column, even though all the numbers are positive. Also, remember that the minimum number of digits to the left of the decimal point is one, since numbers with no integer part will still print with a single leading zero.

For SCI display format, use a skip index of 3 plus the number of desired extra spaces to the left of the sign position. Since all entries in scientific notation contain only one digit to the left of the decimal point, there is only one skip index corresponding to this format.

For ENG display format, CP does not automatically align the decimal points. If the skip index is 3, then any number will be placed immediately after the previous entry in the print buffer, regardless of whether the mantissa is 1, 2 or 3 digits long. One must, therefore, use an index of 3, 4 or 5, depending whether the mantissa is 3, 2 or 1 digit long, respectively. In addition, if the largest mantissa is only 2 rather than 3, then skip indexes would be 3 for 2 digit mantissas and 4 for 1 digit ones. Likewise, if all mantissas are 1 digit long, then a skip index of 3 for all entries will suffice. In a program, a check for this could be to test the length of the exponent. The correct skip index would be:

For maximum mantissas = 3 digits:
 Index = 5 - (Exponent MOD 3) + # extra spc.
For maximum mantissas = 2 digits:
 Index = 4 - (Exponent MOD 3) + # extra spc.
For maximum mantissas = 1 digit:
 Index = 3 + # extra spaces
even though all the numbers are positive.

In our first example, we have column 1 with no commas, FIX 2 and 3 digits left of the decimal point. From table 1 this gives us a skip index of 2. likewise, the indexes for columns 2 and 3 would be 1 and 4 respectively. For column 4, which is ENG 0 with a maximum of 1 mantissa digit, the index would be 3. The keystroke sequence to create table 1 would then be:

Keystrokes	Display	Result
FIX 2	0.00	Set display mode
CF29	0.00	and commas off
2 STO 06	2.00	Store skip index
3.21	3.21	1st value into
XEQ CP	3.21	print buffer
FIX 0	3	2nd display mode
1 STO 06	1	2nd skip index
2	2	2nd value into
XEQ CP	2	print buffer
FIX 1	2.0	3rd display mode
SF29	2.0	and commas on
4 STO 06	4.0	3rd skip index

1304.5	1,304.5	3rd value into
XEQ CP	1,304.5	print buffer
ENG 0	1. 03	4th display mode
CF29	1. 03	and commas off
3 STO 06	3. 00	4th skip index
3 EEX 6 CHS	3. -06	4th value into
XEQ CP	3. -06	print buffer
PRBUF	3. -06	Prints last line

The above keystroke sequence would print the first line of table 1. To print the other lines of the table, one would follow the sequence for the first line, except to substitute the line 2 values 3.26, 8, 6814.3 and 1 EEX +30 for line 1 values 3.21, 8, 1304.5 and 3 EEX -6, then line 3 values, and finally the last set of values.

```
  3.21  2 1,304.5 3.-06
 43.26  8 6,814.3 1.+30
  0.58 10 1,313.1 6.-09
618.18  1 4,441.6 3.-12
```

MORE EXAMPLES OF CP

Example 2. Print the information in table 3 on the 82143A printer using the CP routine:

ROM PERIPHERAL ROUTINES:

NAME	BYTES	DEVICE	SIZE
LG	45	PRINTER	0
HS	40	PRINTER	6
HA	50	PRINTER	6
CP	60	PRINTER	7
BA	337	WAND	19
MP / HP	596	PRINTER	35

Table 3. Information to be printed using the CP routine for example 2

Here is a case where we must have both ALPHA and numeric columns in the same printed lines. The length of the ALPHA information is not consistent down the two ALPHA columns, so there should be a way that the 41C can know how to left justify the ALPHA entries. Below is a routine, written by Ron Yankowski (2980) which left justifies ALPHA entries.

ALPHA Column Print Formatting: This routine will left-justify data in the ALPHA register and accumulate it into the print buffer. If the information is shorter than a user designated length, then spaces will be added to fill the remaining columns. If the ALPHA is too long, the string will be truncated at the designated length. The width may be from 1 to 18 characters. The instructions are listed below:

Keystrokes	Display	Result
N	N	Enter maximum column
STO 07	N	width (18 or less)
ALPHA (text) ALPHA	(text)	Key the text into the ALPHA register
XEQ ACP	(text)	Text is added to the print buffer left justified

The column width value in register 07 remains unchanged after executing ACP, so it does not need to be reloaded if the same column is being left justified repeatedly. The listing of ACP is below:

BAR CODE ON PAGE 479

APPLICATION PROGRAM FOR: CP

Program	Comments
01♦LBL "ACP"	
02 6	
03 RCL 07	
04 "⊢ "	
05 X<=Y?	1 to 6 char's long
06 GTO 14	
07 RCL Y	
08 -	
09 ASTO Z	
10 ASHF	
11 SF 10	
12 "⊢ "	
13 X<=Y?	7 to 12 char's long
14 GTO 14	
15 RCL Y	
16 -	
17 ASTO T	
18 ASHF	
19 SF 09	
20 "⊢ "	Greater than 12 characters long
21♦LBL 14	
22 -	
23 ASTO T	
24 CLA	
25 X>0?	
26 XEQ 13	
27 ARCL T	
28 ASTO X	
29 LASTX	
30 CLA	
31 XEQ 13	
32 ARCL Y	Restoring ALPHA register
33 ASHF	
34 ASTO X	
35 CLA	
36 FS?C 10	
37 ARCL Z	
38 FS?C 09	
39 ARCL T	
40 ARCL X	
41 ACA	
42 RTN	
43♦LBL 13	Append X no. of blanks onto string
44 "⊢ "	
45 DSE X	
46 GTO 13	
47 END	

Routine ACP uses R07 and flags 10 and 09 as well as the stack. It leaves ALPHA intact for later use.

Returning to example 2, we nay now use the ACP routine to create both of the ALPHA columns in the example. Use the column width values of 5 and 7 for ALPHA columns 1 and 2 respectively. The first numeric column is FIX 0 with no commas and 3 maximum digits (skip index = 2 from table 2), and the second numeric column is FIX 0 with 2 maximum digits left of the decimal point. However this second numeric column is an extra 2 characters to the right of the previous one, allowing a position for the sign. Therefore, use 2+2 or 4 digits, yielding a skip index of 3 from table 2. The resulting keystroke sequence is:

KEYSTROKES	DISPLAY	RESULT
ALPHA ROM (space) PERIPHERAL (space ROUTINES: ALPHA	(text)	Enter header
XEQ PRA	(text)	Print line
XEQ ADV		Skip a line
ALPHA (space) NAME (space) BYTES (space SIZE (space) DEVICE ALPHA	(text)	Header
XEQ PRA	(text)	Print header
FIX 0		Set display mode
5 STO 07	5	1st ALPHA column
ALPHA LG ALPHA	LG	ALPHA entry
XEQ ACP	LG	Left justifies
2 STO 06	2	1st skip index
45 XEQ CP	45	Add to buffer
1 SKPCHR	1	Skip a space
7 STO 07	7	2nd ALPHA column
ALPHA PRINTER ALPHA	(text)	ALPHA entry
XEQ ACP	(text)	Left justifies
4 STO 06	4	2nd skip index
0 XEQ CP	0	Add to buffer
XEQ PRBUF	(text)	Prints buffer
5 STO 07	5	1st ALPHA column
ALPHA HS ALPHA	HS	ALPHA entry
XEQ ACP	HS	Left justifies
2 STO 06	2	1st skip index
40 XEQ CP	40	Add to buffer
7 STO 07	7	2nd ALPHA column
ALPHA PRINTER ALPHA	(text)	ALPHA entry
XEQ ACP	(text)	Left justifies
4 STO 06	4	2nd skip index
6 XEQ CP	6	Add to buffer
XEQ PRBUF	6	Prints buffer

etc.
(Continues for lines 3 to 6 similarly.)

The Printer Preparation Form.

In order to better prepare printer outputs for column alignment, a form has been provided which allows composition of the full 24-character lines for determination of CP skip indexes. Along with the printer columns, the format of each column may be included, for easier programming. Remember that for columns which will be aligned by CP, an extra space must be allotted for a sign position, whether one is present or not. This is because CP uses function ACX, which leaves room for the sign before the number. Since one would usually leave a space between columns anyway, this is not a problem. However, if an extra space is inserted, then 2 spaces will appear if all the numbers in the column are positive.

Two copies of the preparation form are included. The first is filled out for the two previous examples. The other is blank, and should be photocopied for use in preparing future outputs requiring CP .

Automatic Multiple Numeric Column Formatting:

Routine CPP is one which automates the formatting of multiple columns of all-numeric information. It can also be used before or after ALPHA columns have been placed in the buffer, leaving a string of consecutive numeric columns to be added. The instructions are shown below:

Load the data registers with the information required for the first line of the table:

```
      R08 = bbb.eee where bbb=first reg. of data
                          eee=last reg. of data
   R(bbb) = 1st column numeric value
 R(bbb+1) = ±a.bc where a = skip index for 1st column
                        b:  1=FIX, 2=SCI, 3=ENG
                        c = # display digits (0 to 9)
            +a.bc = CF29 (no commas), -a.bc = SF29 (commas)
 R(bbb+2) = 2nd column value
 R(bbb+3) = ±a.bc for second column
    .           .
    .           .
    .           .
 R(eee)   = nth column value
 R(eee+1) = ±a.bc for nth column
```

Place any ALPHA information in the buffer, then XEQ CPP. Now add trailing ALPHA if any, and PRBUF and the line is printed. The procedure for each successive line of the printed table is: Accumulate columns into the buffer, load data registers, XEQ CPP for the string of consecutive numeric columns, add any other columns to the buffer, then PRBUF. The listing of CPP is below:

BAR CODE ON PAGE 479

APPLICATION PROGRAM FOR: CP

```
01♦LBL "CPP"
02 CF 29
03 2 E-5                  Set counter in R08
04 ST+ 08
05♦LBL 00
06 RCL 08
07 1
08 +
09 RCL IND X              Recall next register
10 X<0?
11 SF 29                  Test for commas
12 ENTER↑
13 INT
14 ABS
15 STO 06                 Store skip index
16 RDN
17 FRC
18 10
19 *
20 ENTER↑
21 INT
22 X<>Y
23 FRC
24 10
25 *
26 X<>Y
27 1
28 X=Y?
29 FIX IND Z              Testing for
30 RDN                    specified
31 2                      display mode
32 X=Y?
33 SCI IND Z
34 RDN
```

PRINTER PREPARATION FORM FOR COLUMN PRINT FORMATTING

Table from Example 1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
			3	.	2	1			2		1	,	3	0	4	.	5		3	.	−	0	6
		4	3	.	2	6			8		6	,	8	1	4	.	3		1	.	+	3	0
			0	.	5	8		1	0		1	,	3	1	3	.	1		6	.	−	0	9
	6	1	8	.	1	8			1		4	,	4	4	1	.	6		3	.	−	1	2

	COLUMN 1	COLUMN 2	COLUMN 3	COLUMN 4	COLUMN 5	COLUMN 6	COLUMN 7	COLUMN 8
DISPLAY MODE	FIX 2	FIX 0	FIX 1	ENG 0				
COMMAS? (Y/N)	NO	NO	YES	NO				
DIGITS LEFT OF DEC. PT.	3	2	4	1				
SKIP INDEX	2	1	4	3				

Table from Example 2.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
N	A	M	E		B	Y	T	E	S		D	E	V	I	C	E		S	I	Z	E		
L	G						4	5		P	R	I	N	T	E	R					0		
H	S						4	0		P	R	I	N	T	E	R					6		
H	A						5	0		P	R	I	N	T	E	R					6		
C	P						6	0		P	R	I	N	T	E	R					7		
B	A					3	3	7		W	A	N	D							1	8		
M	P	/	H	P		5	9	6		P	R	I	N	T	E	R				3	5		

	COLUMN 1	COLUMN 2	COLUMN 3	COLUMN 4	COLUMN 5	COLUMN 6	COLUMN 7	COLUMN 8
DISPLAY MODE	ALPHA	FIX 0	ALPHA	FIX 0				
COMMAS? (Y/N)		NO		NO				
DIGITS LEFT OF DEC. PT.	5 CHAR'S	3	7 CHAR'S	2				
SKIP INDEX		2		4				

PRINTER PREPARATION FORM FOR COLUMN PRINT FORMATTING

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

	COLUMN 1	COLUMN 2	COLUMN 3	COLUMN 4	COLUMN 5	COLUMN 6	COLUMN 7	COLUMN 8
DISPLAY MODE								
COMMAS? (Y/N)								
DIGITS LEFT OF DEC. PT.								
SKIP INDEX								

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

	COLUMN 1	COLUMN 2	COLUMN 3	COLUMN 4	COLUMN 5	COLUMN 6	COLUMN 7	COLUMN 8
DISPLAY MODE								
COMMAS? (Y/N)								
DIGITS LEFT OF DEC. PT.								
SKIP INDEX								

Program	Comment
35 3	
36 X=Y?	
37 ENG IND Z	
38 RCL IND 08	Recall numeric value
39 XROM "CP"	Call to CP
40 ISG 08	
41 GTO 00	Increment counter
42 END	

The barcode for CPP appears in Appendix N. After scanning, insert 'SF 29' after 01 LBL "CPP". This line was not present in the barcode version of the program.

Example 3. Print the following table of information using CP. The data may be generated in a program to compute square, cube, and fourth roots of the numbers 30 through 50 in increments of 5.

TYPE #	SQU. ROOT	CUBE ROOT	4TH ROOT
A 30	5.477	3.1072	2.3403
B 35	5.916	3.2711	2.4323
C 40	6.325	3.4200	2.5149
D 45	6.708	3.5569	2.5900
E 50	7.071	3.6840	2.6591

Table 4. Data from Example 3 to be added to the print buffer and printed using CP.

Following the single character ALPHA column, there are 4 consecutive columns of numeric information. Therefore, for each line we may load a series of data registers and XEQ CPP to fill the print buffer. Also, since the ALPHA column is consecutive alphabetic characters, we may use ROM routine DC, decimal to character, to generate the letters A through E:

APPLICATION PROGRAM FOR: CP

Program	Comment
01♦LBL "TABLE"	
02 30	
03 STO 07	1st column counter
04 65.069	
05 STO 05	Character counter
06♦LBL 00	
07 CLA	
08 9.015	R09-R15 contains info. for 4 columns
09 STO 08	
10 RCL 07	
11 STO 09	
12 1.1	
13 STO 10	1st column info.
14 RDN	
15 SQRT	
16 STO 11	
17 .13	
18 STO 12	2nd column info.
19 RCL 07	
20 3	
21 1/X	
22 Y↑X	
23 STO 13	
24 .14	
25 STO 14	3rd column info.
26 RCL 07	
27 .25	
28 Y↑X	
29 STO 15	
30 .14	
31 STO 16	4th column info.
32 RCL 05	
33 XROM "DC"	Call to DC
34 ACA	
35 XEQ "CPP"	Call to CP
36 PRBUF	Print buffer
37 5	
38 ST+ 07	Increment 1st column counter
39 ISG 05	
40 GTO 00	
41 END	

```
A 30 5.477 3.1072 2.3403
B 35 5.916 3.2711 2.4323
C 40 6.325 3.4200 2.5149
D 45 6.708 3.5569 2.5900
E 50 7.071 3.6840 2.6591
```

FURTHER DISCUSSION OF CP

Routine CP may be used to create single columns of numbers for the purpose of X-axis labelling in bar charts created by routines HA and HS or in function plots created by MP or HP. Remember, however, that CP requires the use of R06. If this register is also required for storage in the plotting or charting routine, then one must save the original R06 value in another location. It may later be restored to R06 after CP has been executed. See the writeups on these other PPC ROM routines elsewhere in this manual.

Another application of CP is to create 'extended' tables of numeric and ALPHA information which are wider than the printer paper. This can easily be done by building the tables 24 columns at a time. Using this technique, one can create tables of virtually any dimensions. Simply attach the printer paper strips along side one another to complete the table.

Routine Listing For: CP

```
76♦LBL "CP"      96♦LBL 04
77 RND           97 -
78 RCL 06        98 SKPCHR
79 RCL Y         99 RDN
80 ABS          100 ACX
81 X≠0?         101 RTN
82 LOG          102♦LBL 06
83 INT          103 R↑
84 9            104 R↑
85 FS? 40       105 .5
86 X<Y?         106 FC? 29
87 GTO 06       107 RND
88 RDN          108 INT
89 X<0?         109 3
90 CLX          110 +
91 FC? 29       111 GTO 04
92 GTO 04       112 END
93 .75
94 /
95 INT
```

LINE BY LINE ANALYSIS OF CP

Line 77 rounds the numeric value to the constraints of the preset display format.

Lines 78 to 98 compute the number of spaces which must be skipped between the most recently occupied buffer position and the current numeric value to be accumulated.

Lines 99 to 101 add the numeric value to the print buffer.

Lines 102 to 111 add additional information to the space-skipping calculation based on the status of flags 40 and 29.

REFERENCES FOR CP

See PPC Calculator Journal, V7N5P8 and V7N10P11.

CONTRIBUTORS HISTORY FOR CP

This routine, originally written by William Cheseman (4381), accumulated a single ALPHA, plus a single numeric column of information into the print buffer when executed. Through suggestions and work by Bill Hermanson (415), Roger Hill (4940), Nicholas Peros (2392) and Jack Sutton (5622), the routine was improved to aid the user in accumulating multiple numeric columns into the print buffer. Had there been additional space in the ROM, we would have attempted to incorporate the ALPHA-justifying and multiple-column formatting routines, presented in the discussion above.

FINAL REMARKS FOR CP

There are many aspects of CP with which the user is recommended to experiment. For instance, if a numeric value in a FIX-decimal formatted column is smaller than the smallest possible value that can be represented in that format, the value is printed as zero. This keeps additional columns aligned in the print buffer. Had CP allowed this 'underflow' value to be printed in SCI format, there would be no guarantee that the columns would remain aligned. If the actual value is desired in this column, the user is reminded by the zero value to increase the number of decimal digits designated for that numeric column.

If a numeric value in a column should overflow to SCI format, CP will adjust the character skip-value so that the number is printed right-justified in the column. This will keep the additional columns aligned.

FURTHER ASSISTANCE ON CP

Contact Jake Schwartz (1820) at 7700 Fairfield St., Phila., Penna. 19152 (home phone 215-331-5324); or Roger Hill (4940) at 300 S. Main St., Apt 5, Edwardsville, Ill. 62025 (home phone 618-656-8825).

TECHNICAL DETAILS

XROM: 20,27 — **CP** — SIZE: 007

Stack Usage: ALL USED
- 0 T:
- 1 Z:
- 2 Y:
- 3 X:
- 4 L:

Alpha Register Usage: NONE USED
- 5 M:
- 6 N:
- 7 O:
- 8 P:

Other Status Registers: NONE USED
- 9 Q:
- 10 ⊢:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

ΣREG: NOT USED

Data Registers:

R00:	NOT USED
R06:	SKIP INDEX
R07:	NOT USED
R08:	NOT USED
R09:	NOT USED
R10:	NOT USED
R11:	NOT USED
R12:	NOT USED

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: AND

Angular Mode: NOT USED

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: 04, 06

Execution Time: 2 seconds

Peripherals Required: 82143A Printer

Interruptible? YES
Execute Anytime? NO
Program File: LG
Bytes In RAM: 60
Registers To Copy: 29

Other Comments: This routine loads the print buffer, but does not PRBUF.

APPENDIX B ROM PROJECT CONTRIBUTORS

NOTE: P - Programming D - Documentation O - Other: Specific Task

Akimi, Kiyoshi (3456)	P, O
Albillo, Valentin (4747)	P
Allen, Charles (4691)	P, D, O: Testing
Altman, Barry (4555)	D
Anonymous	P, D, O
Bachlund, Gary (1399)	O: Cover Artwork
Barnes, Ed (1004)	D, O: Tab Labels, Wrap
Barnette, Bill (1514)	P, D
Barsabu, Eric (4304)	P
Beimesch, Wayne (5854)	O
Bell, Joe (5781)	O; Wrap
Bercovitz, Nat (4694)	D
Bertucelli, Harry (3994)	P, D, O: Proofreading
Borkman, Leigh (5218)	P
Borman, Roy (3933)	O: Packaging Engineer
Browning Dan (5052)	SDS Testing
Buck, Carter (4783)	P, D
Burkhart, John (4382)	P
Cadwalader, Tom (3502)	P, D, O
Carrie, Clif (834)	P, O
Camacho, Alejo,(Lee) (5843)	O
Castelli, Eddie (5393)	O: Donated Mag Cards
Cheeseman, William (4381)	P, D
Close, Charles (3878)	P
Collett, Richard (4523)	P, O
Cullings, Steve (192)	O
Dearing, John (2791)	P, O
DeArras, Jim (4706)	P, O
Dennes, Graeme (1757)	P, D
Dewey, Don (5148)	P, D
Doig, Jon (4318)	P
Donaldson, George (3825)	D
Duba, George (4248)	D, O: Black Th. Paper
Eldridge, George (5575)	P
Evans, Ray (4928)	P, D
Fauser, Doug (4968)	O
Fischer, Tim (5793)	P
Fraundorf, P (1025)	P
Gibbs, Ernest (4610)	O: Special Characters
Gordon, Ron (3449)	P
Groom, Robert (5127)	O: Sb discovery
Habegger, Janet (2305½)	O: Wrap
Habegger, Richard (2305)	O: Wrap
Hall, Richard H. (4803)	P, D
Harris, Charlie (1959)	O: Bar Codes
Helman Den (3059)	O
Hermanson, Bill (4115)	P
Heyman, Vic (850)	P
Hill, Roger (4940)	P, D, O: See Dedication
Hiser, James (4352)	D
Hooper, Tom (1769)	D
Horn, Jim (1402)	P
Ingram, Emmett (17)	O: Serial No.'s, Wrap
Jacobs, Steven R. (5358)	P
Jarett, Keith (4360)	ROM Committee
Kaplan, David (3678)	P, D
Kaslow, David (1725)	D, O: Proofreading
Keith, David (5825)	P
Kendall, Keith (5425)	O
Kennedy, John (918)	ROM Committee
Killian, Chuck (4163)	O: Pasteup
Knapp, Ron (618)	P
Kolb, Bill (265)	P
Kuenning, Geoff (5071)	O
Kuyt, Frits (236)	P, O: Labels
Latham, Mark (1748)	O: Tone Table
Lavins, Lawrence (7310)	D
Lee, Jerry (5504)	O: Donated Mag Cards
Lew, Clinton (5578)	P
Lilly Terry (5080)	O: Testing
Lind, Paul (6157)	P, D, O
Linick, Evan (7023)	D
Lopez Lopez, Fernando (2887)	P, D, Proofreading
Malaga, Ernie (6594)	O: Bar Code Verify
Malm, Don (1362)	P
Matson, Les (5608)	P, D
McCurdy, Greg (3957)	D
McGechie, John (3324)	P, D, O
Meyer, Nathan (4795)	P, O
Meyers, Harvey (3101)	P
Murdock, Bruce (2916)	P, D, O: Order Sort
Nelson, Richard (1)	ROM Committee
Noble, Richard (9)	O: Order Audit
Pearce, Craig (311)	P
Peros, Nicholas (2392)	P
Pickard, Bill (3514)	O: Sb Discovery
Pratt, Randall (2860)	O
Price, Barry (4146)	O: Calligraphy
Predmore, Read (5184)	P, D
Ragsdale, Charles (7251)	D
Reinstein, Cary (2046)	P
Rosenfield, Carl (5591)	P
Roussel, Phillipe (4367)	P
Schwartz, Jake (1820)	ROM Committee
Schwartz, Richard (2289)	P, D, O: Pasteup, Wrap
Sitte, Martin (6224)	P, D
Slocum, Charles (2907)	D, O: Donated Paper
Spear, David (5488)	P, D
Stern, Clifford (4516)	P, D, O: Proofreading
Strobele, Cal (1502)	O: Pagination
Stout, Jack (1221)	D
Sutton, Jack (5622)	P, D, O: Donated Mag Cards
Tenzer, Gary M.(1816)	P, D, O
Trebing, Mark (4421)	D, O: Word Processing
Trinh, Phi (6171)	P
Uphues, Hans (5286)	P, O
Vaughan Robert (739)	O: Timing
Vogel, Lee (4196)	O: Pasteup
Wada, Bob (3234)	O: Wrap, Order Audit
Wandzura, Steve (4635)	P, O
Weinstein, Iram (6051)	P, D
Weisenburger, Larry (1793)	P
Westen, Gerard (4780)	P
Wheeler, Fred (1150)	P, O
White, David (5353)	D, O
Wickes, William (3735)	P, D, O: Intro. Syn. Prog.
Wimsatt, William (5807)	P, O
Yankowski, Ron (2980)	P, O
Zarum, David (4736)	P

END

CU - CURTAIN UP

CU moves the curtain up or down according to the contents of X. Typically, an integer (positive or negative) is entered, but if X is not an integer, only the integer part plays a role.

BACKGROUND FOR CU

See Appendix M on Curtain Moving.

Example 1: The sequence 12, XEQ CU raises the curtain by 12 registers, so that R_N has become R_{N-12} for N ≥ 12. The former R00 thru R11 reside below the curtain as explained in Appendix M on Curtain Moving.

Example 2: Using CU and HN instead of LB.

LB is a nice, refined way to put bytes anywhere you want them. However, if you have a large number of bytes to enter there is a brute force method that is faster. This is to enter a register at a time instead of a byte at a time. The disadvantage is that all bytes entered with this method go to the beginning of program memory. As an example, entry of the "goose" program is shown. (For more information on the "goose", see PPCJ, V7N5P55.)

1. Write down the bytes you need to enter. Here we will enter two instructions and nine text lines. Ensure that they are in order.

9C	0A							(Fix A)
F7	01	00	00/	00	00	C0	13	(text)
F7	01	00/	00	00	0C	00	13	(text)
F7	01/	00	00	00	C0	00	13	(text)
F7/	01	00	00	0C	00	00	13	(text)
F7	01	00	00	C0	00	00/	13	(text)
F7	01	00	0C	00	00/	00	13	(text)
F7	01	00	C0	00/	00	00	13	(text)
F7	01	0C	00/	00	00	00	13	(text)
F7	01	C0/	00	00	00	00	13	(text)
CE	75							(X<>M)

2. Starting from the last byte, divide them into seven byte groups. This comes out as eleven groups. Ensure that SIZE is equal or greater than eleven and no important data is in the first eleven data registers.

3. In alpha, enter the last group, in this case 13CE75. (HN adds leading zeros, all trailing zeros must be entered!) XEQ HN ; STO 00. (For ease, assign HN to any key.) Enter the next-to-last group (13F701C0); XEQ HN ; STO 01.

4. Continue entering, converting and storing until the last group is stored in R10.

5. Enter 11; XEQ CU . This moves the curtain up past the 11 registers holding the instructions putting them in program memory. To get to these instructions CAT 1; R/S immediately; GTO .000. The instructions will be here in order. If you have made a mistake and program memory starts to get scrambled, the best way to recover is: 11 CHS XEQ CU . This moves the instructions back into data memory. Remember that STO does not normalize, but you cannot recall to see where a mistake was made as RCL does normalize. To get the rest of "goose" entered GTO .000 and key in instructions as below. When you get to a synthetic instruction, just SST past the instruction already in memory. While I prefer this method for entering bulk instructions, LB is still the best to put synthetic instructions exactly where they are needed.

```
01♦LBL "GOOSE"      11 XEQ 99           21 XEQ 99
 02 FIX 0           12 "x♦♦μ♦♦Ȧ"        22 "x♦♦♦♦Ȧ"
 03 CF 21           13 XEQ 99           23 XEQ 99
 04 CF 28           14 "x♦♦♦♦Ȧ"         24 STOP
 05 CF 29           15 XEQ 99
 06 "x♦♦♦♦Ȧ"        16 "x♦μ♦♦♦Ȧ"        25♦LBL 99
 07 XEQ 99          17 XEQ 99           26 X<> [
 08 "x♦♦♦μ♦Ȧ"       18 "x♦♦♦♦Ȧ"         27 VIEW X
 09 XEQ 99          19 XEQ 99           28 RTN
 10 "x♦♦♦♦Ȧ"        20 "xμ♦♦♦♦Ȧ"        29 .END.
```

COMPLETE INSTRUCTIONS FOR CU

X, XEQ CU raises the curtain X registers. If X is negative the curtain is lowered. The new curtain location C = C? + x must be within the range 1 ≤ C ≤ 16 or 193 ≤ C ≤ 256 + n * 64, where n is the number of single density memory modules present. Invalid inputs give MEMORY LOST. The former Y and Z are preserved in X and Y. The old c register ends up in T and alpha is cleared.

Although CU can be used in a program, generally HD / UD or (or ΣC) would be preferable since they are much faster. For manual operation (with no before or after restrictions) CU is very handy. Should a program entailing curtain manipulations be stopped before completing execution, CU offers an easy path to recovering access to all data registers: merely XEQ S? , subtract the original SIZE, and XEQ CU

If CU is used in a program to "save" (make inaccessible) some data registers before calling a subroutine, and then to "restore" (make accessible again) these registers when control is returned to the calling program, the proper calling sequence is

```
    :               XEQ  subroutine
    n                    -n
 XROM CU            XROM CU
                          :
```

Appendix M on "CURTAIN MOVING" presents an overview of curtain manipulations. CU can also be used to convert non-normalized codes in data registers to program steps (see, for example, PPC CJ, V7N6P42c. ΣC cannot be used for this purpose, because it normalizes two of the registers (the former R00 and R05).

CU is not always interruptible with the printer present. To make it interruptible, download it and replace line 136 by "⊦---A". This ensures that the printer will not find flag 55 clear and set flag 21, possibly (50% chance) altering the .END. pointer. CU is interruptible with the printer attached if, and only if 4 ≤ (E? mod 8) ≤ 7. If (E? mod 8) ≥ 4 then increase or decrease SIZE by 4 (see ΣC). CU does not alter the pointer to the statistical register block. Therefore, the (relative) ΣREG location will be decreased by X, and may even be negative, meaning that the statistical registers lie below the new curtain.

MORE EXAMPLES OF CU

Example 3: Suppose you want to use the curve fit program CV , but you only have enough registers available for SIZE 021. Since CV uses R06 through R26 you would appear to be out of luck. However, if you don't need to use R00 through R05 you can key in -6 XEQ **CU** and the HP-41 will have SIZE 027. The first six data registers now contain part of your first program in catalog 1, so don't use R00 through R05 or that program. You can use **CV** normally now, but don't forget to key in 6 XEQ **CU** after you're done with **CV**. This restores the original SIZE 021 curtain.

Example 4: Raise the curtain over all data registers using the two step sequence XEQ **S?** , XEQ **CU**. If you CLRG before and PACK afterward, you have virtually the equivalent of executing SIZE 000.

Routine Listing For: **CU**

```
131♦LBL "CU"
132 ABS
133 RDN
134 RCL c
135 STO [
136 "⊢♦♦♦♦"
137 11
138 X<> [
139 X<> d
140 STO ]

141♦LBL 00
142 RDN
143 X<> L
144 INT
145 X=0?
146 GTO 14
147 2
148 /
149 RCL [
150 X<>Y
151 FRC
152 X=0?
153 GTO 13

154♦LBL 01
155 FC?C IND Y
156 SF IND Y
157 FC? IND Y
158 CHS
159 X>0?
160 GTO 13
161 FC? IND Y
162 CHS
163 DSE Y
164 GTO 01

165♦LBL 13
166 DSE [
167 GTO 00

168♦LBL 14
169 X<> ]
170 X<> d
171 STO [
172 "⊢ABC"
173 X<> \
174 X<> c
175 RDN
176 CLA
177 RTN
```

LINE BY LINE ANALYSIS OF CU

Regard the initial contents of status register c as the following 14 hex digits:

$$s_1s_2s_30\ 01\ 69\ z_1z_2\ z_3e_1\ e_2e_3$$

where: $s_1s_2s_3$ = the abs. 3 hex-digit address of the first Σ-register

$z_1z_2z_3$ = the abs. 3 hex-digit address of R00 (i.e., the curtain pointer)

$e_1e_2e_3$ = the abs. 3 hex-digit address of reg. containing .END.

Line 132 transfers X to register L. Lines 133 and 142 serve to retain as much of the stack as is practical. Together with line 175 they serve to ensure that Y and Z at the time of entry to **CU** are retained as X and Y, respectively, upon exit.

Lines 135 through 137 place the contents of status register c in alpha register so as to overlap registers N and M as follows:

N	M
$00\ 00\ 00\ \ s_1s_2\ s_30\ 01\ 69$	$z_1z_2\ z_3e_1\ e_2e_3\ 00\ 00\ 00\ 00$

Lines 137 through 140 transfer the contents of register M to flag register d to perform a bit-by-bit addition of the contents of L to the absolute register address of R00 now corresponding to flags 00 through 11. Additionally, the original contents of register d are saved in register O and a flag counter in register M is initialized to 11.

Lines 141-164 constitute the binary addition loop. Lines 143-144 result in the integer part of the contents of L replacing the contents of X. If zero, addition has been completed (lines 145-146). Otherwise divide by two (lines 147-148), fetch current contents of register M (line 149), and interchange stack registers X and Y (line 150), so that the quotient (of the division by 2) is in the X-register. Line 151 results in transferring this quotient to register L, and places the fractional part of this quotient (0, +½, or -½) in register X. If zero, proceed to the next bit (lines 152-153).

The computing loop of lines 154-164 adds to (or subtracts from) the sum (or difference) being accumulated in flag register d, a single non-zero binary digit (a '1') of the curtain movement argument (passed to **CU** in register X upon entry). Successive bits (least-significant to most-significant) of this argument are evaluated by repeated divisions by 2 of the integer part of the previous quotient. This loop (lines 154-164) is entered only when a quotient is not an integer (corresponding bit is a '1'). Register M is marking the bit location (in flag register d) where the '1' is being added (or subtracted).

The process is quite straightforward. Consider addition first. Upon entering the loop, 0.5 is in the X-register and register Y points to the bit position (flag in register d) of the added '1'. (recall that at line 149 this was initialized by the contents of register M). The bit (flag) is toggled in lines 155-156, and at line 157, the FC? command is checking whether the bit is now a zero. If so, we change the sign of the contents of X. When adding, this yields a negative number, so we remain in the loop (lines 159-160) and restore the positive number (lines 161-162), decrement the pointer in Y (move 1 bit to the left) and loop (lines 163-164). When the flipped bit is now a '1', we leave this inner loop to adjust the pointer in M stepping the outer loop (lines 141-164).

Subtraction is no less straightforward. Upon entering the loop, -0.5 is in the X-register. Thus, we return to the outer loop when the flipped bit is now a zero, and stay in the inner loop when the flipped bit is now a '1'.

Once the addition (or subtraction) is completed (test and branch at lines 145-146), we proceed to restore status register d (lines 169-170), place

$$z_1'z_2'\ z_3'e_1\ e_2e_3\ x_1x_2\ x_3x_4\ x_5x_6\ x_7x_8$$

into register M (lines 170-171), where $\overline{z_1'z_2'z_3'}$ = $\overline{z_1z_2z_3}$ + entry contents of X, and $x_1x_2x_3x_4x_5x_6x_7x_8$ are "don't care" hex-digit values, shift the alpha register left 3 bytes in order to fetch and place the new contents fo status register c (lines 172-174). Lines 175-176 clear the scratch work from the alpha register and return the X and Y registers to their state just prior to 'pushing' the curtain movement parameter onto the stack to call **CU**.

TECHNICAL DETAILS

XROM: 10,34	CU	SIZE: 000

Stack Usage: 0 T: new c 1 Z: CHANGED 2 Y: Z 3 X: Y 4 L: used	**Flag Usage:** MANY USED BUT ALL RESTORED 04: 05: 06: 07: 08:
Alpha Register Usage: 5 M: 6 N: 7 O: ALL CLEARED 8 P:	09: 10: 25:
Other Status Registers: 9 Q: NOT USED 10 ⊢: NOT USED 11 a: NOT USED 12 b: NOT USED 13 c: ALTERED 14 d: USED BUT RESTORED 15 e: NOT USED	**Display Mode:** UNCHANGED **Angular Mode:** UNCHANGED **Unused Subroutine Levels:** 6
ΣREG: REDUCED BY X **Data Registers:** NONE USED R00: R06: R07: R08: R09: R10: R11: R12:	**Global Labels Called:** Direct: NONE Secondary: NONE **Local Labels In This Routine:** 00 01 13 14
Execution Time: 0.7 seconds to 5.0 seconds.	
Peripherals Required: NONE	
Interruptible? ONLY IF PRINTER NOT ATTACHED* Execute Anytime? NO! Program File: VM Bytes In RAM: 85 Registers To Copy: 60	**Other Comments:** Improper inputs can give MEMORY LOST. *If printer is attached, Flag 21 will be set. This will alter the .END. pointer unless 4 ≤(E? mod 8)≤ 7.

REFERENCES FOR CU

Introductory articles on curtain moving can be found in the *PPC CALCULATOR JOURNAL*, (V7N4P23, V7N5P45).

CONTRIBUTORS HISTORY FOR CU

Curtain moving was first proposed by Bill Wickes (3735) in PPC CJ, V6N8P27d. In December 1979, Harry Bertuccelli (3994) wrote a long program using the original CODE AND DECODE programs to implement programmable curtain moving. It worked, but it was very slow. A relatively fast curtain control program, "CR" (see PPC CJ, V7N4P23), was written independently by Keith Jarett (4360). It used binary-decimal-binary conversion by flag manipulation. Bill Wickes then wrote CU, using direct binary addition to save bytes and execution time.

FINAL REMARKS FOR CU

CU may be the oldest routine in the PPC ROM. That it has stood the test of time is a tribute to its author.

FURTHER ASSISTANCE ON CU

Call William C. Wickes (3735) at (503) 754-0117.
Call Keith Jarett (4360) at (213) 374-2583.

NOTES

APPENDIX C ROM ROUTINE AUTHOR LIST

A few of the routines contained in the PPC CUSTOM ROM were written entirely by one person. The vast majority, however, were written by several PPC members either together or as code segments assembled by the ROM committee member responsible for the group the routine belonged to. This list is an attempt to give author credit to a specific individual. In some cases the contribution was equal and two authors are listed. The list is provided in two forms. The second is in routine name order, the first is in author order.

Author	Routines
Albillo, Valentin (4747)	DT
Buck, Carter (4783)	NC RF SU
Cadwallader, Tom (3502)	Ab Sb VK
Cheeseman, William (4381)	AL CP
Dennes, Graeme (1757)	FI
Dewey, Don (5148)	FI
Eldridge, George (5575)	BD TB
Evans, Ray (4928)	S2 S3
Fischer, Tim (5793)	HP MP
Gordon, R. (3449)	HS
Hill, Roger (4940)	+K B 1K 2D A? BI CB CD CJ CK CP DC DP E? F? HN IF JC L LB LF LG LR MK NH OM PA PD PK QR RK RT Rb TN VA VF VM VS XD XE XL
Jarett, Keith (4360)	AM BL CX DS EP FL GE HD IP MA ML PS
Jarett, Keith Cont'd.	RD RX S? SD SK SX UD Σ? ΣC
Kaplan, Dave (3678)	EX MT
Kennedy, John (918)	BC BM BR BΣ CA CM CV DF DR FR GN IR M1 M2 M3 M4 M5 PM PR SE SV
Lind, Paul (6156)	LR SR
Malm, Don (1362)	RN
Meyers, Harvey (3101)	S1
Nelson, Richard (1)	BA MS SM PO
Phi, Trinh (6171)	NP
Predmore, Read (5184)	IG
Reinstein, Cary (2046)	T1
Schwartz, Jake (1820)	HA
Schwartz, Richard (2289)	BE BV BX FD
Stern, Clifford (4516)	C? NR NS
Westen, Gerard (4780)	AD
Wickes, William (3735)	CU DT

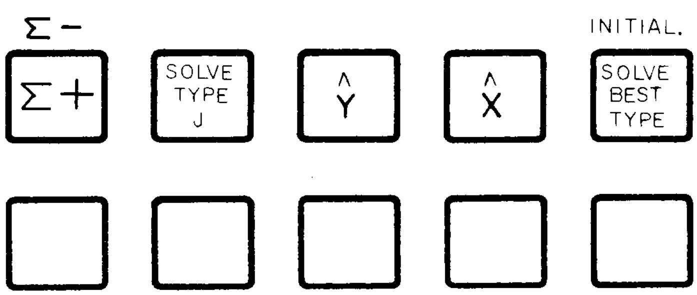
Routine	Author
+K	Hill, Roger (4940)
B	Hill, Roger (4940)
1K	Hill, Roger (4940)
2D	Hill, Roger (4940)
A?	Hill, Roger (4940)
AD	Westen, Gerard (4780)
AL	Cheeseman, William (4381)
AM	Jarett, Keith (4360)
Ab	Cadwallader, Tom (3502)
BA	Nelson, Richard (1)
BC	Kennedy, John (918)
BD	Eldridge, George (5575)
BE	Schwartz, Richard (2289)
BI	Hill, Roger (4940)
BL	Jarett, Keith (4360)
BM	Kennedy, John (918)
BR	Kennedy, John (918)
BV	Schwartz, Richard (2289)
BX	Schwartz, Richard (2289)
BΣ	Kennedy, John (918)
C?	Stern, Clifford (4516)
CA	Kennedy, John (918)
CB	Hill, Roger (4940)
CD	Hill, Roger (4940)
CJ	Hill, Roger (4940)
CK	Hill, Roger (4940)
CM	Kennedy, John (918)
CP	Cheeseman, William (4381)
	Hill, Roger (4940)
CU	Wickes, William (3735)
CV	Kennedy, John (918)
CX	Jarett, Keith (4360)
DC	Hill, Roger (4940)
DF	Kennedy, John (918)
DP	Hill, Roger (4940)
DR	Kennedy, John (918)
DS	Jarett, Keith (4360)
DT	Wickes, Willaim (3735)
	Albillo, Valentin (4747)
E?	Hill, Roger (4940)
EP	Jarett, Keith (4360)
EX	Kaplan, Dave (3678)
F?	Hill, Roger (4940)
FD	Schwartz, Richard (2289)
FI	Dennes, Graeme (1757)
	Dewey, Don (5148)
FL	Jarett, Keith (4360)
FR	Kennedy, John (918)
GE	Jarett, Keith (4360)
	Hill, Roger (4940)
GN	Kennedy, John (918)
HA	Schwartz, Jake (1820)
HD	Jarett, Keith (4360)
HN	Hill, Roger (4940)
HP	Fischer, Tim (5793)
HS	Gordon, R. (3449)
IF	Hill, Roger (4940)
IG	Predmore, Read (5184)
IP	Jarett, Keith (4360)
	Hill, Roger (4940)
IR	Kennedy, John (918)
JC	Hill, Roger (4940)
L	Hill, Roger (4940)
LB	Hill, Roger (4940)
LF	Hill, Roger (4940)
LG	Hill, Roger (4940)
LR	Lind, Paul (6156)
	Hill, Roger (4940)
M1	Kennedy, John (918)
M2	Kennedy, John (918)
M3	Kennedy, John (918)
M4	Kennedy, John (918)
M5	Kennedy, John (918)
MA	Jarett, Keith (4360)
MK	Hill, Roger (4940)
ML	Jarett, Keith (4360)
MP	Fischer, Tim (5793)
MS	Nelson, Richard (1)
MT	Kaplan, Dave (3678)
NC	Buck, Carter (4783)
NH	Hill, Roger (4940)
NP	Phi, Trinh (6171)
NR	Stern, Clifford (4516)
NS	Stern, Clifford (4516)
OM	Hill, Roger (4940)
PA	Hill, Roger (4940)
PD	Hill, Roger (4940)
PK	Hill, Roger (4940)
PM	Kennedy, John (918)
PO	Nelson, Richard (1)
PR	Kennedy, John (918)
PS	Jarett, Keith (4360)
	Hill, Roger (4940)
QR	Hill, Roger (4940)
RD	Jarett, Keith (4360)
RF	Buck, Carter (4783)
RK	Hill, Roger (4940)
RN	Malm, Don (1362)
RT	Hill, Roger (4940)
RX	Jarett, Keith (4360)
Rb	Hill, Roger (4940)
S1	Meyers, Harvey (3101)
S2	Evans, Ray (4928)
S3	Evans, Ray (4928)
S?	Jarett, Keith (4360)
SD	Jarett, Keith (4360)
SE	Kennedy, John (918)
SK	Jarett, Keith (4360)
SM	Nelson, Richard (1)
SR	Lind, Paul (6156)
	Hill, Roger (4940)
SU	Buck, Carter (4783)
SV	Kennedy, John (918)
SX	Jarett, Keith (4360)
Sb	Cadwallader, Tom (3502)
T1	Reinstein, Cary (2046)
TB	Eldridge, George (5575)
TN	Hill, Roger (4940)
UD	Jarett, Keith (4360)
UR	Kennedy, John (918)
VA	Hill, Roger (4940)
VF	Hill, Roger (4940)
VK	Cadwallader, Tom (3502)
VM	Hill, Roger (4940)
VS	Hill, Roger (4940)
XD	Hill, Roger (4940)
XE	Hill, Roger (4940)
XL	Hill, Roger (4940)
Σ?	Jarett, Keith (4360)
ΣC	Jarett, Keith (4360)

CV - CURVE FIT

This program will determine a curve of best fit to a set of data points. The four standard curve types the program handles are:

1. Linear $y = b*x + a$
2. Exponential $y = a*e^{bx}$ (a>0)
3. Logarithmic $y = b*Ln(x) + a$
4. Power $y = a*x^b$ (a>0)

The program will compute the coefficients a and b in the equation of one of the above four curve types as well as compute a value r^2 called the coefficient of determination which is a measure of the goodness of fit. Once a set of data has been fit to a given curve type, a prediction may be made for the y-value given a new x-value, or a prediction may be made for the x-value given a new y-value. The functions available on the top row of keys on the keyboard are indicated in the following diagram.



These same functions are referenced in the examples and instructions by enclosing the name of the function on the key in square brackets [ ].

Example 1: Find the straight line which best fits the following data:
(1.1, 5.2), (4.5, 12.6), (8.0, 20.0),
(10.0, 23.0), (15.6, 34.0)
Then predict y when x=20 and predict x when y=25.

Plug the PPC ROM into the 41C and SIZE 027. GTO "CV" and go into USER mode. This puts the program counter in ROM and makes the curve fit functions available on the top row of keys. Pressing [INITIALIZE] will initialize the program. This clears registers R11 thru R24 so that a new set of data may be entered. In this example the 5 data points will be entered using the [Σ+] key. Key in each pair as x ENTER↑ y and push [Σ+].

Do:		See:
[INITIALIZE]		1.0000
1.1 ENTER↑ 5.2	[Σ+]	2.0000
4.5 ENTER↑ 12.6	[Σ+]	3.0000
8.0 ENTER↑ 20.0	[Σ+]	4.0000
10.0 ENTER↑ 23.0	[Σ+]	5.0000
15.6 ENTER↑ 34.0	[Σ+]	6.0000

All the data has now been entered and the parameters for the curve will be computed next. Since in this example we are interested in a straight line we key 1 (j=1) and push [SOLVE TYPE j]. When execution stops the values a, b, and r are available in the stack as:

Z:	r	and are also stored as	R08:	b
Y:	a		R09:	a
X:	b		R10:	r

For this example:

Z: r=0.999035140.
Y: a=3.499147270
X: b=1.972047542

The value r ranges between -1 and +1 and is a measure of how well the data fits the given curve type. The sign of r indicates whether the data is positively or negatively skewed. The closer r is to one of the extremes ±1 the better the fit. For this example the line has positive slope and the fit is extremely good (all sample problems seem to work well).

Having computed the values b and a (these remain stored in R08 & R09 until new data is input) we can determine new points along the line. Key in 20 and push [$\hat{y}$] for the predicted y-value. y=42.94009811 when x=20. Key in 25 and push [$\hat{x}$] for the predicted x-value. x=10.90280649 when y=25.

COMPLETE INSTRUCTIONS FOR CV

(Keyboard Operation)

1) Key GTO "CV", SIZE 027 and go into USER mode. The keyboard functions should now be now available on the top row of keys.

2) Press [INITIALIZE] to initialize the program. This step clears data registers R11 thru R24 inclusive. These registers will be used to accumulate the data for all four curve types. The display will show 1.

3) Key in the next data pair (x,y) as x ENTER↑ y and push [Σ+]. Repeat this step for all data pairs. The display will stop with a count of the number of the next data pair to be entered. This feature makes it possible to enter only the y-values when the x-values are consecutive integers which start counting from 1. In this case the display provides the x-values which need not be entered. If an improper data pair has just been input with the [Σ+] key, then immediately pressing R/S will delete the pair. Otherwise an improper or undesired data pair can be deleted by re-entering both x and y and pressing [Σ-].

4) As data pairs are entered it is possible that some x or y value is negative or zero. In these cases only one or two of the four curve types may be applied to the data. The four curve types and their respective equations are as follows:

Type J	Name	Equation
1	Linear	$y = b*x + a$
2	Exponential	$y = a*e^{bx}$ (a>0)
3	Logarithmic	$y = b*Ln(x) + a$
4	Power	$y = a*x^b$ (a>0)

If any x-values are negative or zero then only types 1 & 2 are feasible curves. If any y-values are negative or zero then only types 1 & 3 are feasible curves. If in any data pair both x and y are negative or zero then type 1 is the only feasible curve. The a coefficient must be positive for curve types 2 and 4.

5) After all data pairs have been input the next step is to select the desired curve type. This step can be accomplished in one of two ways. Under either option, the 41C should not be interrupted or else there is a possibility that the data registers will not be returned with their normal contents.

a) To fit a particular curve type, key in the number 1-4 for that type and press [SOLVE TYPE j]. The stack returns with:

Z: r	and these parameters	R07: j=curve type
Y: a	remain stored in	R08: b
X: b		R09: a
		R10: r

Step a) may be repeated at any time for any of the four curve types.

b) If all data input is positive then pressing [SOLVE BEST] will automatically choose the curve of best fit according to the curve type with largest absolute value of r. In this case the stack returns with:

T: r	and these parameters	R07: j=curve type
Z: a	remain stored in	R08: b
Y: b		R09: a
X: j=best curve type		R10: r

6) Predictions for new x or y values may be made only after step 5) has been completed. Predictions for new values are based on the settings of flags F08 and F09 which are automatically set during the fit process in step 5). The status of flags 8 and 9 for the four curve types are as follows.

		Flag 8	Flag 9
1	Linear	clear	clear
2	Exponential	set	clear
3	Logarithmic	clear	set
4	Power	set	set

In general the user need not be concerned with these flag settings, and F08 and F09 are not avalible for other use and must not be disturbed. To predict y given x, key in x and press [$\hat{y}$]. To predict x given y, key in y and press [$\hat{x}$]. In both cases the predicted value is left in the X-register.

7) New data may be added or deleted at any time via the [Σ+] or [Σ-] keys. However, step 5) must be performed after updating the data before any new predictions can be made using step 6). The parameters a and b are automatically destroyed after input of new data.

MORE EXAMPLES OF CV

Example 2: Determine whether the following data points are better suited for a logarithmic curve or a power curve. Then re-input the same x values and see how close the program predicts the y values.
(8, 2), (27, 3), (40, 3.2), (50, 3.5), (100, 4.1)

Do:	See:
[INITIALIZE]	1.0000
8 ENTER↑ 2 [Σ+]	2.0000
27 ENTER↑ 3 [Σ+]	3.0000
40 ENTER↑ 3.2 [Σ+]	4.0000
50 ENTER↑ 3.5 [Σ+]	5.0000
100 ENTER↑ 4.1 [Σ+]	6.0000

We will now try to fit a logarithmic curve type 3. Key 3 [SOLVE TYPE j]. The program returns with:

Z: 0.997148866 = r
Y: 0.267411352 = a
X: 0.822629796 = b

We next try to fit a power curve which is type 4. Key 4 [SOLVE TYPE j]. The program returns with:

Z: 0.995179948 = r
Y: 1.127479133 = a
X: 0.285458085 = b

Choosing the best r we would assume a type 3 logarithmic curve with the equation:

y = (0.822629796)*Ln(x) + 0.267411352

Since we just finished the power curve fit, the power curve parameters are still in the machine and hence we must go back and key 3 [SOLVE TYPE j] to return to the logarithmic parameters. Now we can predict the y's using the original x's. The predicted y-values are shown to four decimal places.

Do:	See:
8 [$\hat{y}$]	1.9780
27 [$\hat{y}$]	2.9787
40 [$\hat{y}$]	3.3020
50 [$\hat{y}$]	3.4856
100 [$\hat{y}$]	4.0558

Example 3: The following data fits either a linear or exponential curve. Determine which is more appropriate. (2, 12), (-1, 2), (3, 17), (5, 23) Then predict y when x = -10. After solving the above problem add the following as additional data points and resolve the same problem.
(-4, 0.713), (2.5, 10.93), (6, 47.53), (10, 254.95)

Do:	See:
[INITIALIZE]	1.0000
2 ENTER↑ 12 [Σ+]	2.0000
1 CHS ENTER↑ 2 [Σ+]	3.0000
3 ENTER↑ 17 [Σ+]	4.0000
5 ENTER↑ 23 [Σ+]	5.0000

Note that since one of the data points has a negative x the only possible curves to be fit under this program are linear or exponential. For a linear fit key 1 [SOLVE TYPE j]. The program returns:

Z: 0.997577939 = r
Y: 5.520000000 = a
X: 3.546666667 = b

For an exponential fit key 2 [SOLVE TYPE j]. The program returns:

Z: 0.958629344 = r
Y: 3.826163699 = a
X: 0.419923419 = b

Choosing the best r we find a linear fit is more appropriate. Since we just finished the exponential fit, the exponential parameters are still in the machine and hence we must go back and key 1 [SOLVE TYPE j] to return the linear parameters. Now key 10 CHS [$\hat{y}$] to predict y = -29.94666667 when x = -10.

We next add the additional data points and resolve the problem. (Do not clear the original data). The display should show 6 after entering the first new data pair below.

Do:	See:
4 CHS ENTER↑ 0.713 [Σ+]	6.0000
2.5 ENTER↑ 10.93 [Σ+]	7.0000
6 ENTER↑ 47.53 [Σ+]	8.0000
10 ENTER↑ 254.95 [Σ+]	9.0000

For a new linear fit key 1 [SOLVE TYPE j]. The data returned is:

Z: 0.765698771 = r
Y: 0.978958100 = a
X: 15.33154618 = b

For a new exponential fit key 2 [SOLVE TYPE j]. The data returned is:

Z: 0.993615263 = r
Y: 3.825595338 = a
X: 0.419945301 = b

Now choosing the best r we see that the new data reflects a change in the curve type. Since the exponential parameters should still be in the machine we can predict y when x = -10. Key 10 CHS [ŷ]. y = 0.057398396.

Example 4: Fit the best curve to the following set of data points.
(1, 2), (2, 2.828), (3, 3.464), (4, 4), (5, 4.472), (6, 4.899), (7, 5.292), (8, 5.657), (9, 6).

In this example the x-coordinates start counting from 1 and are consecutive integers. So we need only input the y-coordinates, but they must be in the proper order. The count in the display will serve as the x-coordinates.

Do:	See:
[INITIALIZE]	1.0000
2 [Σ+]	2.0000
2.828 [Σ+]	3.0000
3.464 [Σ+]	4.0000
4 [Σ+]	5.0000
4.472 [Σ+]	6.0000
4.899 [Σ+]	7.0000
5.292 [Σ+]	8.0000
5.657 [Σ+]	9.0000
6 [Σ+]	10.0000

Since all the data are positive we may use the best fit function to let the program find the best fit among all 4 curve types. Press [SOLVE BEST]. The contents of the stack when the program stops are:

T: 0.999999994 = r
Z: 1.999855865 = a
Y: 0.500043886 = b
X: 4.000000000 = best curve type

This indicates a power curve (type 4) where the equation is of the form:

$y = (2.00)*x^{0.50}$ (values rounded to 2 places)

APPLICATION PROGRAM 1 FOR CV

Curve fit solutions are often more meaningful when the points input are also plotted, superimposed on the plot of the "best fit" or selected equation type. The CVPL program will function exactly as CV functions and, after calculating the parameters a, b, r and r↑2, the program will stop with the prompt: "TO PLOT: R/S" To plot the equation calculated with the points input superimposed, simply press the R/S key. Nothing else need be done to obtain a plot. When accomplished in this way, the default situation, all numbers will be printed with 2 decimal places and the resulting plot will contain 50 plotted points. The detailed instructions include options to print other than 2 decimal places and to plot a smaller or greater number of points. The same key captions used by CV are used, plus the shifted keys b, c, and d for the optional features indicated.

Note that this program can also be used without the printer and will function essentially the same as CV but with the display labeling the points entered, showing deletions indentified as such, and labeling the parameters calculated.

The plotting program takes into account all possibilities: duplicate, identical points; almost identical points that would plot as identical; points with identical x-values but with significantly different y-values; individual single points. Any quantity of duplicate points can be handled. The points are plotted with 4 plotting characters as follows:

a. The equation of type J is plotted using a small square dot (box). One equation point is normally plotted before the first input point and after the last point. If the first input point is close to zero, it will be plotted first.

b. Individual single points are plotted with a large X.

c. Two (or more) essentially identical points are plotted with a double X, two small x's, one above the other.

d. Two (or more) points having essentially the same x-value but having different y-values are plotted with an asterisk located where the largest of the point's (based on x-value) plot should be. If desired the other points not shown for this value of x could be drawn in by hand or more points could be selected for the plot to separate very close x-values.

To simplify the program and reduce the number of registers required to store the data points, both the x- and y-value of a point are stored in one register, using a decimal point to separate them. This limits the magnitude and sign of the numbers to the following: data points must be nonzero, positive numbers and less than 1000 in magnitude. If you need to deal with larger numbers, shift all decimal points before entering them. Note: if the program is used without the printer, or by pressing "NO PLOT" with a printer, none of these restrictions apply and the "data error" message will not be encountered if you try to use negative or large numbers. See the valid use of negative entries in the CV instructions, however.

This program was developed originally as a modification to Gary Tenzer's curve fit program, "CFIT" in the PPC JOURNAL, V7N5P46, and was to be published in the JOURNAL as a stand alone program. The program was 691 steps in length (1414) bytes. With the CV routine (plus others such as the S2 sorting routine) the plotting routine was completely re-written to utilize as many of the ROM routines as possible and the end result is presented below, significantly improved over my earlier version, with 439 steps and using 865 bytes (8 tracks on 4 cards). Most of Gary's displays and labeling are used in this program which partially account for the length of the program. I feel these extras are desirable, especially when using a printer.

Example 1 for CVPL: Use the same problem as Example 1 for CV. Find the linear equation for the following data: (1.1, 5.2), (4.5, 12.6), (8.0, 20.0), (10.0, 23.0), (15.6, 34.0).
Then predict y when x=20 and predict x when y=25.

Plug the PPC ROM in and using the card reader, read in all 8 sides of the program CVPL. Put the calculator in USER mode. Connect printer and put in MAN mode. Press initialize (shift E) and the display will tell you to SIZE 038 plus the number of points you plan to input. For this example SIZE 043 (=38 + 5 points). Press R/S to complete intialization of the program. See 1.00 in the display asking for the first point's values. First however, we will select 4 decimal places in the printout so key in 4 and press shift C (for the number of decimal places). See 1.00 again asking for the first point's values. Key in each point exactly as in the CV instructions by keying in X ENTER↑ Y [Σ+]. Keyboard functions assigned to keys are shown in square braces [ ] below.

Do:	See:
[Initialize]	"SIZE=38+ PTS"
XEQ "SIZE" 043	0.00 (size=38+5)
R/S	1.00 TONE 9
4 [No.Dec.Places]	1.00 TONE 9
1.1 ENTER↑ 5.2 [Σ+]	X1=1.0000, Y1=5.2000 2.0000 TONE 9
4.5 ENTER↑ 12.6 [Σ+]	X2=4.5000, Y2=12.6000 3.0000 TONE 9
8.0 ENTER↑ 20.0 [Σ+]	X3=8.0000, Y3=20.0000 4.0000 TONE 9
10.0 ENTER↑ 23.0 [Σ+]	X4=10.0000, Y4=23.0000 5.0000 TONE 9
15.6 ENTER↑ 34.0 [Σ+]	X5=15.6000, Y5=34.0000 6.0000 TONE 9

Since we want a linear curve, we key in 1 and push [SOLVE TYPE J]. When execution stops the following will be printed.

```
1: LIN
a=3.4991
b=1.9720
r=0.9990
r↑2=0.9981
```

In the calculator display see "TO PLOT: R/S"
If we now press R/S the plot will consist of 50 points. To select plot of 25 points, key in 25 and press [No.Pts.In Plot], the shifted B key. The same display will appear in the calculator (nothing is printed). Before plotting, we will first find the predicted y and x values asked for. Key in 20 and push [$\hat{y}$], the C key. Printed (and displayed) see:

"IF X = 20.0000, Y = 42.9401"

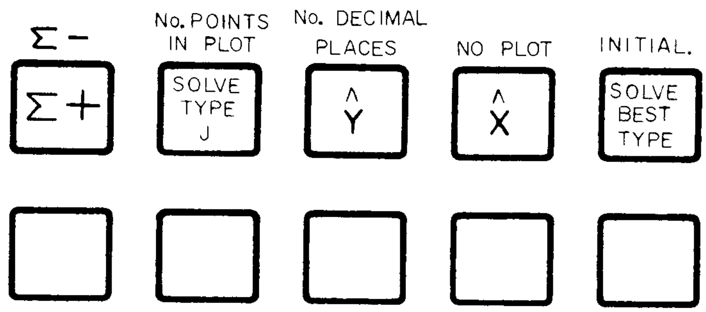
Key in 25 and push [$\hat{x}$], the D key and see:

"IF Y = 25.0000, X = 10.9028"

Now press R/S to plot the data. When the plotting is complete, wait for the BEEP before stopping the calculator.

The total time for this example, except for sizing the calculator was 4 min. and 25 sec. The primary consumer of time is normally the plotting, so the number of points selected greatly effects execution time. Often a short plot of 15 points is adequate.

After the BEEP has sounded the completion of the plot you can find other predicted values of x or y, select a different curve type, add points or delete points and see the effect on the new plot.



INSTRUCTIONS	INPUT DATA	KEYS	OUTPUT
1. Load cards, sides 1-8 in USER mode			0.00
2. Initialize		shift E	"SIZE=38+PTS"
3. If SIZE inadequate Otherwise go to step 4		XEQ "SIZE"	
4. Complete initialization		R/S	TONE 9 1.00
5. Optional - To print without plotting (including negative or larger numbers)		shift D	1.00
(Note: for new problem with plotting, must CF 24)			
6. Optional - Select no. of decimal places to be printed. Default is 2. Or key in n	n	shift C	TONE 9 1.00
7. Enter data point	X ENTER↑ Y	[Σ+]	Xj=--.---- Yj=--.---- TONE 9 # next point
Note: where x-values are same as displayed # of next point, input only Y and press A			
8. If point input is correct go to step 9. If incorrect, press R/S to delete the point just entered.		R/S	"**DELETE**"
To delete any previously entered point, re-enter exact X & Y values and press			X=--.---- Y=--.---- TONE 9 # next point

[Σ-] (same)

9. For each new point wait for TONE 9 and repeat step 7. (same as 7)

Note: Program will accept only positive values of X and Y in the range .01-999.99. For numbers outside of range shift decimal before entering. For a zero value use .01. "DATA ERROR" message will be displayed after an invalid entry. This note only applies with printer connected. Any values for X and Y will be accepted without a printer or after pressing "NO PLOT" with a printer. See CV instructions regarding acceptable negative numbers.

10. Calculate a,b,r,r^2:

a. For "best fit" based on largest ABS value of r: E (typical)

(Note: r & r^2 display correctly only on printer. Final caption not shown if printer not connected)

"1:LIN"
"a=--.----"
"b=--.----"
"r=--.----"
"r↑2=--.----"
"TO PLOT: R/S"

b. For selected type "j" curve
Case:
1: Linear — 1 B
2: Exponential — 2 B (same)
3: Logarithmic — 3 B
4: Power — 4 B

NOTE: Step 10 must be accomplished after all data points have been entered before steps 11, 12, or 13 may be attempted.

11. Optional: select number of points to be plotted (points input plus equation points).
a. Default value = 50 points no action required.
b. Enter # of desired points

n shift B "TO PLOT: R/S"

12. Project y given x — $\hat{x}$ C — "IF X = --.----"
"Y = --.----"
"TO PLOT: R/S"

13. Project x given y — $\hat{y}$ D — "IF Y = --.----"
"X = --.----"
"TO PLOT: R/S"

14. To add additional points to same data, go to step 7.

15. Plot curve and data points — R/S — Curve and points plotted

The following symbols are used:

- ▪ points on curve type "j"
- × data points, no duplicate X or Y value
- ⁑ 2 or more data points with the same X and Y values within the plotting tolerance.
- ✱ 2 or more data points with same X-value but different Y-values. Only one of the points is plotted.

BEEP sounds after plot is complete

Note: after plotting wait for BEEP. Then you can add more points, delete points, predict new X or Y values, plot with a different number of points, calculate curve parameters with a different number of decimal places displayed or select a different curve type by going back to the above instructions.

Example 2 for CVPL: This example will demonstrate all four plotting characters described above and show how deletions and points can be added. The initial points are the following:

(70.00, 11.10), (10.40, 71.86), (22.30, 38.71), (10.50, 73.12), (40.90, 21.73), (4.20, 85.20) (100.30, 1.34), (41.30, 34.70)

Print with 4 decimal places and solve for the best fit curve. Then find the predicted value of y for X=35 and the predicted value of X for Y=100. Then plot using 30 points in the plot. Size for one additional point to be added. In the following the data in parentheses are not printed.

Do:	See:
[initialize]	("SIZE=38+PTS")
XEQ "SIZE" 047	(0.00)
R/S to complete initialization	(1.00 TONE 9)
4 [# dec. places]	(1.00 TONE 9)
70 ENTER↑ 11.1 [Σ+]	"X1=70.0000" "Y1=11.1000" (2.0000 TONE 9)
10.4 ENTER↑ 71.86 [Σ+]	"X2=10.4000" "Y2=71.8600" (3.0000 TONE 9)
22.3 ENTER↑ 38.71 [Σ+]	"X3=22.3000" "Y3=38.7100" (4.0000 TONE 9)
10.5 ENTER↑ 73.12 [Σ+]	"X4=10.5000" "Y4=73.1200" (5.0000 TONE 9)
40.9 ENTER↑ 21.63 [Σ+]	"X5=40.9000" "Y5=21.6300" (6.0000 TONE 9)

Y5 was entered in ERROR so to delete:

R/S	"DELETE" "X=40.9000" "Y=21.6300" (5.0000 TONE 9)

Now continue entering the correct values

40.9 ENTER↑ 21.73 [Σ+]	"X5=40.9000" "Y5=21.7300" (6.0000 TONE 9)
4.2 ENTER↑ 85.2 [Σ+]	"X6=4.2000" "Y6=85.2000" (7.0000 TONE 9)
100.3 ENTER↑ 1.34 [Σ+]	"X7=100.3000" "Y7=1.3400" (8.0000 TONE 9)
41.3 ENTER↑ 34.7 [Σ+]	"X8=41.3000" "Y8=34.7000" (9.0000 TONE 9)

Now push E for [SOLVE BEST]

"3: LOG"
"a=132.4456"
"b=-28.2822"
"r=-0.9812"
"r↑2=0.9627"
("TO PLOT: R/S")

Find the predicted values:

35 [$\hat{y}$] "IF X=35.0000"
"Y=31.8925"
100 [$\hat{x}$] ("TO PLOT: R/S")
"IF Y=100.0000"
"X=3.1494"
("TO PLOT: R/S")

Now select a 30 point plot:

30 [# points in plot] ("TO PLOT: R/S")
R/S to plot the data

After the BEEP sounds and the plotting is complete, add an additional point (71.1, 11.0), almost the same as point 1, and delete what appears to be the worst fitting point (22.30, 38.71).

71.1 ENTER↑ 11.0 [Σ+] "X9=71.1000"
"Y9=11.0000"
(10.0000 TONE 9)

22.30 ENTER↑ 38.71 [Σ-]
** DELETE **
"X=22.3000"
"Y=38.7100"
(10.0000 TONE 9)

Now again solve for the best fit.

[SOLVE BEST] "3: LOG"
"a=133.8645"
"b=-28.5171"
"r=-0.9858"
"r↑2=0.9719"
("TO PLOT: R/S")

We have slightly improved the fit to a log curve and the parameters a and b have of course changed. Now make a new plot by pressing R/S. After replotting the data, again find the predicted values of y if x=35 and x if y=100.

35 [$\hat{y}$] "IF X=35.0000"
"Y=32.4761"
100 [$\hat{x}$] "IF Y=100.0000"
"X=3.2789"

Looking at the plot, note the value of having the first input point be preceded by a point on the LOG curve. Note the double x at x=11 representing 2 almost identical points X2 and X4. The asterisk at x=41 means 2 or more points have essentially the same x-value but very different y-values. They are X5 and X8 and because X8 has a larger x-value than X5, the asterisk is plotted for Y8.

LINE BY LINE ANALYSIS OF CVPL

Lines 02-11 set up default conditions for 50 point plot and 2 decimal place printout. Lines 14-21 display next point to be input. Lines 22-30 are the delete routine using R/S. Lines 31-70 are the delete routine for later deletion of a point which first combines x and y in a single number as YYYYY.XXXXX after rounding to 2 decimal places, then searches stored points registers for the same point. When the point is found a copy of the last point stored is made in that register. Flag F05 prevents display of point number for a delete. Input of new points are added to CV statistical registers (71-118), then x and y values are checked for sign and magnitude and rounded to 2 decimal places and stored in YYYYY.XXXXX format. Lines 86-186 recall full numbers (not rounded) from CV for printing to number of decimal places selected and printout is formatted for input points, deleted x and y, and calculated parameters a, b, r, and r↑2. Lines 187-192 display plotting prompt "TO PLOT: R/S" only if printer connected, so program can be used without printer. Lines 193-200 store the barcoded input plotting symbols. Lines 201-217 exchange registers R07-11 with R33-37 using BE so data needed for CV statistical registers will be saved for later use, not lost when "PRPLOT" in printer ROM uses registers R07-11. Lines 218-236 use BX to find maximum and minimum y values of input points, then increase maximum and decrease minimum y by 25% of range to allow for equation points to be plotted outside of range of input points. Lines 237-241 make Ymin=0 if this value would have become negative after the 25% adjustment. These lines also determine the y-plotting increment used to see if 2 points have essentially same y-value. Lines 244-258 store "CRV" as the curve name for PRPLOT. The next function performed is a reverse of the left and right sides of the decimal point. Points are now stored as XXXXX.YYYYY (244) and S2 is used to sort the stored points to find maximum and minimum x and for faster plotting (246). Also calculated is the x-plotting increment using the range of x-values and number of points wanted in plot. If the x-minimum is smaller than plotting increment, lines 259-266 make the 1st point plotted the smallest x-value of the points; otherwise the x-minimum is set so one equation point will be plotted first. X-max made large enough that PRPLOT will never stop plot so one equation point can be plotted after largest x-values of input points (267-275). Stop routine initiated when one equation point beyond last point has been plotted. Lines 277-292 restore the statistical registers for CV by XEQ BE , then reverse stored points to original YYYYY.XXXXX format (284). Lines 293-296 reset the counter and "BEEP", ready for changes to data, etc. Flags 02 and 00 are used to determine if plotting is complete, lines 329-330. Routine to check stored points to see if they should be plotted at this x-value (297-323), checks +50% of plotting increment from this plotting point. If flag F03 is set (324) at least one point to be plotted here, and still checking for others. Plotting symbol to be used selected (340-360) and stored in R03 for "PRPLOT" to use for plotting. Where 2 input points have essentially the same x-value, checks to see if their y-values are also essentially the same (361-378). Flag F04 is set when 2 points have the same y-values, F01 is set when they have significantly different y-values (375-377). Plotting routines for the 4 curve types are in steps 379-399. The routine to reverse the left and right sides of the stored points (from the decimal point) is LBL 16, steps 400-419. Storage routines for optional selection of number of points in plot and number of decimal places in printout are in steps 425-435.
NOTE: The BLSPEC numbers for the plotting characters, if barcodes are not used, are:

box:	0, 0, 28, 28, 28, 0, 0
large X:	0, 34, 20, 8, 20, 34, 0
double x:	0, 0, 73, 54, 54, 73, 0
asterisk:	0, 20, 8, 62, 8, 20, 0

The ROM routine BL can also be used to create the equivalent of these BLDSPEC characters.

BAR CODE ON PAGE 481


```
01♦LBL "CVPL"
02♦LBL e
03 4900
04 STO 29
05 2
06 STO 38
07 .
08 "SIZE=38+ PTS"
09 PROMPT
10 STO 06
11 XROM "CV"
12 39.999
13 STO 30
14♦LBL 12
15 RCL 18
16 1
17 +
18 CLA
19 ARCL X
20 TONE 9
21 PROMPT
22 DSE 30
23 SIN
24 SF 10
25 6
26 STO 06
27 RCL 08
28 RCL 09
29 XROM "CV"
30 GTO 08
31♦LBL a
32 SF 10
33 6
34 STO 06
35 RDN
36 XROM "CV"
37 RCL 08
38 RND
39 1 E3
40 /
41 STO 00
42 RCL 09
43 RND
44 1 E2
45 *
46 ST+ 00
47 RCL 30
48 1
49 -
50 STO 27
51 39.999
52 STO 30
53♦LBL 13
54 RCL IND 30
55 RCL 00
56 X=Y?
57 GTO 11
58 ISG 30
59 GTO 13
60♦LBL 11
61 RCL IND 27
62 STO IND 30
63 RCL 27
64 STO 30
65♦LBL 08
66 SF 12
67 "** DELETE **"
68 AVIEW
69 SF 05
70 GTO 09
71♦LBL A
72 1
73 STO 06
74 RDN
75 XROM "CV"
76 RCL 08
77 XEQ 14
78 1 E3
79 /
80 STO IND 30
81 RCL 09
82 XEQ 14
83 1 E2
84 *
85 ST+ IND 30
86♦LBL 09
87 SF 12
88 FIX 0
89 "X"
90 FC? 05
91 ARCL 18
92 FIX IND 38
93 "⊢="
94 ARCL 08
95 AVIEW
96 PSE
97 FIX 0
98 "Y"
99 FC? 05
100 ARCL 18
101 FIX IND 38
102 "⊢="
103 ARCL 09
104 AVIEW
105 PSE
106 ADV
107 FC?C 05
108 ISG 30
109 GTO 12
110♦LBL 14
111 FIX 2
112 999.99
113 X<>Y
114 RND
115 X>0?
116 X>Y?
117 XEQ 17
118 RTN
119♦LBL C
120 SF 03
121♦LBL D
122 FIX IND 38
123 SF 12
124 STO 28
125 3
126 FC? 03
127 4
128 STO 06
129 RCL 28
130 XROM "CV"
131 "IF X="
132 FC? 03
133 "IF Y="
134 ARCL 28
135 AVIEW
136 PSE
137 "Y="
138 FC?C 03
139 "X="
140 ARCL X
141 AVIEW
142 PSE
143 ADV
144 GTO 07
145♦LBL B
146 ENTER↑
147 2
148 GTO 11
149♦LBL E
150 5
151♦LBL 11
152 FIX IND 38
153 SF 12
154 STO 06
155 RDN
156 XROM "CV"
157 "1: LIN"
158 ASTO 01
159 "2: EXP"
160 ASTO 02
161 "3: LOG"
162 ASTO 03
163 "4: PWR"
164 ASTO 04
165 CLA
166 ARCL IND 07
167 AVIEW
168 PSE
169 "a="
170 ARCL 09
171 AVIEW
172 PSE
173 "b="
174 ARCL 08
175 AVIEW
176 PSE
177 "r="
178 ARCL 10
179 AVIEW
180 PSE
181 "r↑2="
182 RCL 10
183 X↑2
184 ARCL X
185 AVIEW
186 ADV
187♦LBL 07
188 FC? 55
189 RTN
190 "TO PLOT: R/S"
191 CF 12
192 PROMPT
193 "♦♦♦♦"
194 ASTO 26
195 "♦α    "
196 ASTO 27
197 "×QAβQ♦"
198 ASTO 28
199 "♦×♦"
200 ASTO 31
201 7.011
202 ENTER↑
203 33.037
204 XROM "BE"
205 RCL 30
206 INT
207 STO Y
208 1
209 -
210 1 E-3
211 *
212 +
213 STO 24
214 FRC
215 39
216 +
217 STO 25
218 XROM "BX"
219 .01
220 *
221 STO 00
222 X<>Y
223 .01
224 *
225 STO 01
226 -
227 ABS
228 .25
229 *
230 ST+ 01
231 ST- 00
232 RCL 00
233 X<0?
234 0
235 STO 00
236 STO 04
237 RCL 01
238 -
239 -62
240 /
241 STO 32
242 "CRV"
243 ASTO 11
244 XEQ 16
245 RCL 25
246 XROM "S2"
247 STO 30
248 RCL 24
249 1
250 -
251 RCL IND X
252 INT
253 RCL 39
254 INT
255 -
256 RCL 29
257 /
258 STO 10
259 RCL 39
260 XEQ 00
261 X<>Y
262 X≠Y?
263 X>Y?
264 -
265 ABS
266 STO 08
267 RCL 24
268 1
269 -
270 RCL IND X
271 XEQ 00
272 3
273 *
274 +
275 STO 09
276 XROM "PRPLOTP"
277♦LBL "CRV"
278 FC?C 00
279 GTO 11
280 7.011
281 ENTER↑
282 33.037
283 XROM "BE"
284 XEQ 16
285 RCL 24
286 INT
287 .999
288 +
289 STO 30
290 FIX IND 38
291 BEEP
292 STOP
293♦LBL 11
294 FS? 02
295 GTO 08
296 STO [
297♦LBL 06
298 RCL IND 30
299 XEQ 00
300 2
301 /
302 RCL [
303 +
304 X>Y?
305 GTO 09
306 FS? 03
307 GTO 10
308 GTO 08
309♦LBL 00
310 INT
311 1 E2
312 /
313 RCL 10
314 RTN
315♦LBL 09
316 X<>Y
317 RCL [
318 RCL 10
319 2
320 /
321 -
322 X<=Y?
323 GTO 11
324 FS? 03
325 GTO 10
326♦LBL 08
327 RCL 31
328 STO 03
329 FS?C 02
330 SF 00
331 RCL [
332 GTO IND 33
333♦LBL 11
334 FS? 03
335 GTO 08
336 SF 03
337 ISG 30
338 GTO 06
339 SF 02
340♦LBL 10
341 1
342 ST- 30
343 CF 03
344 RCL 26
345 FS?C 01
346 GTO 15
347 RCL 27
348 FS?C 04
349 GTO 15
350 RCL 28
351♦LBL 15
352 CF 04
353 STO 03
354 RCL IND 30
355 FRC
356 1 E3
357 *
358 ISG 30
359 RTN
360 RTN
361♦LBL 08
362 1
363 ST- 30
364 RCL IND 30
365 FRC
366 ISG 30
367 RCL IND 30
368 FRC
369 -
370 1 E3
371 *
372 ABS
373 RCL 32
374 ISG 30
375 SF 04
376 X<Y?
377 SF 01
378 GTO 06
379♦LBL 02
380 RCL 34
381 *
382 E↑X
383 RCL 35
384 *
385 RTN
386♦LBL 03
387 LN
388♦LBL 01
389 RCL 34
390 *
391 RCL 35
392 +
393 RTN
394♦LBL 04
395 RCL 34
396 Y↑X
397 RCL 35
398 *
399 RTN
400♦LBL 16
401 RCL 25
402 STO 30
403♦LBL 05
404 RCL IND 30
405 STO Z
406 FRC
407 1 E5
408 *
409 STO Y
410 RCL Z
411 INT
412 1 E5
413 /
414 ST+ Y
415 RDN
416 STO IND 30
417 ISG 30
418 GTO 05
419 RTN
420♦LBL 17
421 FC? 24
422 FC? 55
423 RTN
424 0
425 /
426♦LBL b
427 1
428 -
429 100
430 *
431 STO 29
432 GTO 07
433♦LBL c
434 STO 38
435 GTO 12
436♦LBL d
437 SF 24
438 .END.
```

FORMULAS USED IN CV

Linear (Type 1):

(1) $y = b*x + a$

(2) $Y = B*X + A$ where $Y=y$, $X=x$, $A=a$, $B=b$

(3) $x = (y-a)/b$

Exponential (Type 2):

(4) $y = a*e^{b*x}$ $(a>0,\ y>0)$

(5) $Y = B*X + A$ where $Y=\ln(y)$, $X=x$, $A=\ln(a)$, $B=b$

(6) $x = [\ln(y) - \ln(a)]/b$

Logarithmic (Type 3):

(7) $y = b*\ln(x) + a$ $(x>0)$

(8) $Y = B*X + A$ where $Y=y$, $X=\ln(x)$, $A=a$, $B=b$

(9) $x = e^{[(\ln(y) - \ln(a))/b]}$

Power (Type 4):

(10) $y = a*x^b$ $(a>0,\ x>0,\ y>0)$

(11) $Y = B*X + A$ where $Y=\ln(y)$, $X=\ln(x)$, $A=\ln(a)$, $B=b$

(12) $x = e^{[(\ln(y) - \ln(a))/b]}$

The curve fit program determines the least squares fit for the equation Y = B*X + A.

(13) $B = (\Sigma XY - (\Sigma X)(\Sigma Y)/n)/(\Sigma X^2 - (\Sigma X)^2/n)$

(14) $A = (\Sigma Y - (B)(\Sigma X))/n$

(15) $$r^2 = \frac{(\Sigma XY - \Sigma X \Sigma Y/n)^2}{((\Sigma X^2 - (\Sigma X)^2/n)(\Sigma Y^2 - (\Sigma Y)^2/n))}$$

The standard four curve type equations (1), (4), (7), and (10) are all special cases of Y = B*X + A which is equations (2), (5), (8), and (11). Note the distinction between upper and lower case letters. The user inputs and outputs are always in terms of the lower case letters. For example, the data input consists of pairs of the form (x,y) and the coefficients the program determines are a and b. In all four cases b=B. The upper case letters are the quantities that the program uses to "conceptually" work with all four curve types simultaneously.

Routine Listing For: CV

```
01♦LBL "CV"
02 GTO IND 06
03♦LBL A
04♦LBL 01
05 CF 10
06♦LBL 06
07 STO 09
08 X<>Y
09 STO 08
10 ΣREG 13
11 FC? 10
12 Σ+
13 FS? 10
14 Σ-
15 RDN
16 RCL 08
17 ENTER↑
18 X>0?
19 LN
20 ST* Z
21 RCL 09
22 X>0?
23 LN
24 ST* Z
25 X<>Y
26 ΣREG 19
27 FC? 10
28 Σ+
29 FS? 10
30 Σ-
31 R↑
32 FS? 10
33 CHS
34 ST+ 12
35 R↑
36 FS? 10
37 CHS
38 ST+ 11
39 X<> Z
40 SIGN
41 ST+ L
42 RCL 08
43 RCL 09
44 X<> L
45 RTN
46 RCL 08
47 RCL 09
48♦LBL a
49 SF 10
50 GTO 06
51♦LBL B
52♦LBL 02
53 CF 08
54 CF 09
55 STO 07
56 2
57 X<Y?
58 SF 09
59 /
60 FRC
61 X=0?
62 SF 08
63 8
64 ST+ 07
65 XEQ IND 07
66 RCL 17
67 RCL 13
68 RCL 15
69 STO 09
70 *
71 RCL 18
72 /
73 -
74 STO 10
75 RCL 14
76 RCL 13
77 X↑2
78 RCL 18
79 /
80 -
81 STO Z
82 /
83 STO 08
84 RCL 13
85 *
86 ST- 09
87 X<>Y
88 RCL 16
89 RCL 15
90 X↑2
91 RCL 18
92 ST/ 09
93 /
94 -
95 *
96 SQRT
97 ST/ 10
98 XEQ IND 07
99 8
100 ST- 07
101 RCL 10
102 RCL 09
103 FS? 08
104 E↑X
105 STO 09
106 RCL 08
107 RTN
108♦LBL 10
109 RCL 11
110 X<> 17
111 STO 11
112♦LBL 13
113 RCL 21
114 X<> 15
115 STO 21
116 RCL 22
117 X<> 16
118 STO 22
119♦LBL 09
120 RTN
121♦LBL 11
122 RCL 12
123 X<> 17
124 STO 12
125♦LBL 14
126 RCL 19
127 X<> 13
128 STO 19
129 RCL 20
130 X<> 14
131 STO 20
132 RTN
133♦LBL 12
134 RCL 23
135 X<> 17
136 STO 23
137 XEQ 14
138 GTO 13
139♦LBL C
140♦LBL 03
141 FS? 09
142 LN
143 RCL 08
144 *
145 RCL 09
146 FS? 08
```

Listing continued on page 118.

Listing continued from page 117.

Routine Listing For: **CV**

```
147 LN
148 +
149 FS? 08
150 E↑X
151 RTN
152♦LBL D
153♦LBL 04
154 FS? 08
155 LN
156 RCL 09
157 FS? 08
158 LN
159 -
160 RCL 08
161 /
162 FS? 09
163 E↑X
164 RTN
165♦LBL e
166♦LBL 00
167 11.024
168 XROM "BC"
169 E
170 RTN
171♦LBL E
172♦LBL 05
173 .
174 STO 25
175 4
176 STO 07
177♦LBL 07
178 RCL 07
179 XEQ B
180 RCL 25
181 RCL 10
182 ABS
183 X<=Y?
184 GTO 15
185 STO 25
186 RCL 07
187 STO 26
188♦LBL 15
189 DSE 07
190 GTO 07
191 RCL 26
192 XEQ 02
193 RCL 26
194 END
```

LINE BY LINE ANALYSIS OF CV

Line 02 provides access to all numeric labels within CV.

Lines 03-45 perform the function of inputting the next data point. This is the "sigma plus" subroutine. All summations are updated when this routine is called. These summations include sums of

x, x^2, y, y^2, xy, $\ln(x)$, $\ln(x)^2$, $\ln(y)$,

$\ln(y)^2$, $\ln(x)\ln(y)$, $x\ln(y)$, $y\ln(x)$.

Lines 48-50 perform the "sigma minus" function for deleting a data pair. Note that flag 10 is set and then a jump is made into the "sigma plus" function.

Lines 51-107 calculate the parameters a, b, and r, for the curve type using formulas (13), (14), and (15). b is stored in R08, a is stored in R09 and r is stored in R10.

Lines 108-138 are a series of intertwined subroutines which are called in the curve fit process. These routines simply perform a series of register exchanges which place the proper sums in the sigma registers for the calculation of the parameters a, b, and r depending on the curve type selected. Since the exchange is performed twice (once in line 65 and once in line 98) all registers are returned to their original state.

Lines 139-151 perform the calculation of the predicted y value using formulas (1), (4), (7), and (10).

Lines 152-164 perform the calculation of the predicted x value using formulas (3), (6), (9), and (12).

Lines 165-170 perform the initialization for the program by clearing the data registers used to accumulate the sums the program requires.

Lines 171-194 perform the function of selecting the best curve type among the four curve types that the program handles.

NUMERIC LABELS/FUNCTIONS IN THE CV PROGRAM

The following list gives a correspondence between numeric labels and subroutines to be called as part of CV programs. To call a subroutine function from one of your own programs, first store the number corresponding to the desired function in data register R06. Then use the instruction XEQ "CV" as part of your program. The execution times for the more significant subroutines are in seconds and are shown in parentheses.

Numeric Label Number in R06	Keyboard Label	Subroutine Function
00	e	Initialize/clear sigma registers (3.2 sec)
01	A	Sigma Plus function (2.2 seconds)
02	B	Solve Type j (1: 2.9 seconds) (2: 3.5 seconds) (3: 3.4 seconds) (4: 4.3 seconds)
03	C	Predict y
04	D	Predict x
05	E	Solve Best Curve Type (15.7 seconds)
06	a	Sigma Minus function provided F10 is set
09	none	provides simple RTN so no register exchange takes place
10	none	exchange register pairs R11&R17, R15&R21, R16&R22
11	none	exchange register pairs R12&R17, R13&R19, R14&R20
12	none	exchange register pairs R23&R17, R13&R19, R14&R20 R15&R21, R16&R22
13	none	exchange register pairs R15&R21, R16&R22
14	none	exchange register pairs R13&R19, R14&R20

Note that to use the Sigma Minus function you must set flag 10 before calling label 06.

Note that labels 09-14 are not represented by functions on the keyboard. They are only internal subroutines within CV which would seem to perform no useful purpose outside of CV. However, they are documented here only because they may provide the truly curious PPC member with some "hidden" functions that they will no doubt find some application for. Those who never read this will never know what they are missing.

REFERENCES FOR CV

1. Gary M. Tenzer (1816) PPC Journal " Curve Fitting Made Easy": I V5N1P29
II V5N3P9
III V5N6P29
IV V6N3P16
V V7N2P20
VI V7N5P46

2. Cuthbert, Daniel and Wood, Fred, "FITTING EQUATIONS TO DATA," John Wiley, New York, 1971

CONTRIBUTORS HISTORY FOR CV

Curve fitting has long been a topic of great interest among PPC members. Gary M. Tenzer (1816) has written several articles introducing this topic and is responsible among others for the development of curve fit programs for the HP-67 and the HP-41C.

The key equation for this program, Y = B*X + A, was taken from a program written by Keith Jarrett (4360). This one equation unifies the four types of curves fit and greatly simplifies previous programs that accomplish the same functions, but less elegantly. John Kennedy (918) did the final coding of the CV program. Bill Barnett (1514) provided CVPL.

FINAL REMARKS FOR CV

CV is not the most powerful curve fit program ever written for a programmable calculator. Limited space did not allow a more comprehensive routine which would fit up to 16 different types of curves. Only the standard 4 types are present in CV. CV can be extended by those wishing to take advantage of its subroutines.

FURTHER ASSISTANCE ON CV

John Kennedy (918) phone: (213) 472-3110

Gary M. Tenzer (1816) phone: (213) 557-8336

NOTES

TECHNICAL DETAILS

XROM: 20, 08 — **CV** — SIZE: 027

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: used CV type
- 09: used CV type
- 10: used Σ- function
- 25: not used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 4

ΣREG: noused

Data Registers:

R00:	not used		
R06:	function call #		
R07:	CV type	R18:	n
R08:	b, x	R19:	Σ ln(x)
R09:	a, y	R20:	$\Sigma \ln(x)^2$
R10:	r	R21:	Σ ln(y)
R11:	Σ x ln(y)	R22:	$\Sigma \ln(y)^2$
R12:	Σ y ln(x)	R23:	Σ ln(x)•ln(y)
R13:	Σ x	R24:	n
R14:	Σx^2	R25:	best r
R15:	Σ y	R26:	best type
R16:	Σy^2		
R17:	Σ xy		

Global Labels Called:

Direct	Secondary
BC	none

Local Labels In This Routine:
A, B, C, D, E, a, e, 00, 01, 02, 03, 04, 05, 06, 07, 09, 10, 11, 12, 13, 14, 15

Execution Time: see NUMERIC LABELS in the CV documentation

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: CV

Bytes In RAM: 292

Registers To Copy: 42

Other Comments:
The subroutine which fits a given curve type should be allowed to run to its full conclusion so that a double register exchange is properly completed

CX - CURTAIN TO ABS LOCATION IN X

CX moves the curtain to the register specified by the decimal address (range 1-16 or 193-512) in stack register X. Should X not be an integer, only the integer part will play a role.

Example 1: The sequence 16, XEQ CX will move the curtain to a position just beyond the status registers (status register e has the decimal address 15). With the curtain in this position, all of user memory is accessible as data registers. In particular, R176 would be the first register used for key assignments. Note, however, that R00 through R175 do not exist for this curtain position.
For another example, see the E? writeup.

BACKGROUND FOR CX

See Appendix M on Curtain Moving.

COMPLETE INSTRUCTIONS FOR CX

X, XEQ CX will place the curtain at absolute address X.

CX should be used with caution. If X-1 addresses a non-existent register, then X, XEQ CX will result in "MEMORY LOST", if executed manually. In a running program, this sequence will not cause a problem provided the curtain is subsequently moved to an acceptable location (where a register exists just below) before execution stops.

The former contents of Y are saved in X and Y. The old c register contents are left in T and ALPHA is cleared.

CONTRIBUTORS HISTORY FOR CX

CX was added to Bill Wickes' (3735) CU by Keith Jarett (4360) early in the ROM development. Somehow it was never removed, even though it did not conform to the ROM guidelines (not enough RAM bytes saved).

Routine Listing For: CX

```
128♦LBL "CX"
129 XROM "C?"
130 -

131♦LBL "CU"
132 ABS
133 RDN
134 RCL c
135 STO [
136 "├♦♦♦♦"
137 11
138 X<> [
139 X<> d
140 STO ]

141♦LBL 00
142 RDN
143 X<> L
144 INT
145 X=0?
146 GTO 14
147 2
148 /
149 RCL [
150 X<>Y
151 FRC
152 X=0?
153 GTO 13

154♦LBL 01
155 FC?C IND Y
156 SF IND Y
157 FC? IND Y
158 CHS
159 X>0?
160 GTO 13
161 FC? IND Y
162 CHS
163 DSE Y
164 GTO 01

165♦LBL 13
166 DSE [
167 GTO 00

168♦LBL 14
169 X<> ]
170 X<> d
171 STO [
172 "├ABC"
173 X<> \
174 X<> c
175 RDN
176 CLA
177 RTN
```

LINE BY LINE ANALYSIS OF CX

Lines 129 through 130 evaluate the appropriate argument for using CU to move the curtain. See CU for an explanation of subsequent processing.

FURTHER ASSISTANCE ON CX

Call William C. Wickes (3735) at (503) 754-0117.
Call Keith Jarett (4360) at (213) 374-2583.

TECHNICAL DETAILS

XROM: 10,33 — **CX** — SIZE: 000

Stack Usage:	Flag Usage: MANY USED BUT ALL RESTORED
0 T: OLD c	04:
1 Z: USED	05:
2 Y: Y	06:
3 X: Y	07:
4 L: USED	08:
Alpha Register Usage:	09:
5 M: ALL CLEARED	10:
6 N: ALL CLEARED	
7 O: ALL CLEARED	
8 P: ALL CLEARED	25:
Other Status Registers:	**Display Mode:** UNCHANGED
9 Q: NOT USED	
10 ├: NOT USED	
11 a: NOT USED	**Angular Mode:** UNCHANGED
12 b: NOT USED	
13 c: ALTERED	
14 d: USED BUT RESTORED	**Unused Subroutine Levels:**
15 e: NOT USED	5
ΣREG: ABSOLUTE VALUE UNCHANGED	**Global Labels Called:** Direct: CU (IN LINE), C? — Secondary: NONE
Data Registers: NONE USED	**Local Labels In This Routine:** 00, 01, 13, 14

Execution Time: 1.7 to 6.0 seconds.

Peripherals Required: NONE

Interruptible? ONLY IF PRINTER NOT ATTACHED*	**Other Comments:**
Execute Anytime? NO	*See Addendum for CU
Program File: VM	
Bytes In RAM: 94	
Registers To Copy: 60	

APPENDIX D REFERENCES & ACCESSORIES (COMMERCIAL PRODUCTS)

The PPC ROM User's Manual is a statement of the state of PPC Applications art. It provides a wide range of information for the HP-41C/CV ueser. Appendix D is a quick reference to various products that are avaliable from a wide varity of sources.

HARDWARE

EPROM BOX- 4/8/16K Switched Capacities,Auto On/Off with calculator,SDS compatible.Dallas Development Systems (214)238-1776. Write 7410 Stillwater,Garland ,TX.75042 U.S.A.

EPROM BOX-Compact De Arras(5.8"x3.6"x1.1") 4/8/16K Switched Capacities,28 pin ZIF Sockets,41 Powered,graceful power downs,SDS compatible;Mike Weaver(704)377-3841 or write FMWA,Inc. 6201 Fair Valley Dr. Charlotte,NC. 28211,U.S.A. ;PPC Member Discount

PORTEXTENDER- Adds 7 slots to 41 calculator,six switchable,one dedicated to printer,lithium power source,(7"x3"x5/8"), AME,Box 373,13450 Maxella,G185,Marina del Rey,CA.90291 U.S.A. (213)306-1249.

SOFTWARE

APPLICATIONS PROGRAMMING-Star Fleet Engineering, 1328 N. Santa Anita Ave., Arcadia CA 91006 USA 213/447-6574

AVIATION PROGRAMS-Dr. T. D. Bolt, 256 Deerwalk Place, Thousand Oaks CA 91320 USA

AVIATION PROGRAMS-Infotec Development Inc., 5402 Bolsa Ave., Huntington Beach CA 92649 714/891-5851

HHC PROFESSIONAL PROGRAMMING-Horizons Technology Inc., 7830 Clairmont Mesa Blvd. San Diego CA 92111 USA 714/292-8331

INCOME TAX PROGRAMS-Touch Button Programs, Inc., 370 Lexington Ave. Room 909, New York

INCOME TAX PROGRAMS- 3rd year, well documented, Anthony A. Vertnro, 2007 Alban Lane, Bowie MD 20716 USA

LA PLACE TRANSFORMS-Raymond D. Morre, PO Box 72, West Covina CA 91793 USA

MISCELLANEOUS

BAR CODE PRODUCTION SERVICE-Georges Lithograph,620 Second Street,San Francisco,CA.94107,U.S.A.(415)392-2400

CALCULATOR BOOKS-Educalc Book Store,27963 Cabot Road,South Laguna,CA.92677,U.S.A.

"CALCULATOR CALCULUS"-Edu-Calc Publications,Box 974,Laguna Beach,CA.92652,U.S.A.

"CALCULATOR TIPS & ROUTINES"-Corvallis Software,Inc.,P.O.Box 1412,Corvallis,OR.97330,U.S.A.

"SYNTHETIC PROGRAMMING"-Larkin Publications,4517 NW Quens Ave.,Corvallis,OR.97330,U.S.A.

BAR CODE SHEETS (SYNTHETIC CODES)-(3)8 1/2x11" sheets of off-set printed codes,264 different bar codes,to load all synthetic states for register M-E. Jacob G. Schwartz,7700 Fairfield St.,Philadelphia,PA.19152,U.S.A. (215) 331-5324

CARRYING CASES, PRINTER MOD ETC.-Phillip Karras, 11821 Idlewood Road, Wheaton MD 20906 USA

CARRYING CASES ,41 SYSTEM,Aluminum Case,Andy L.Burg,Marketing Systems International,18516 Mayall Street,Suite G,Northridge,CA.U.S.A.

COLORED CARDS-Leonard Prince,767 W.Rosslyn Ave.,Fullerton,CA.92632,U.S.A.

MAGNETIC CARD ORGANIZERS-8 1/2x11" clear vinyl pages punched for loose leaf book,66 pockets per page,IMTEC,P.O.Box 1402,Bowie,MA. 20716,U.S.A.

NAPA COMP CARD HOLDERS-Tam's Inc.,14932 Garfield Ave.,Paramount,CA. 90723,U.S.A.(213)633-3262

POCKET HEX TABLE-Same as supplied with the PPC ROM.1st card $3.00; additional cards $1 US,CAN.&MEX.;$1.20 elsware,US orders deduct $1 for SASE (.20) postage per 4 cards.Mail to:Keath Jarett,1540 Mathews Ave.,Mahattan Beach,CA.90266,U.S.A.

RPN-41 Work Station- Desk top device holds 41 calculator,printer and card reader in secure position.Capital Calculator Company,INC.,701 East Gude Drive,Rockville,Maryland 20850,U.S.A.Frank Cohen (301)340-7200

END

DC - DECIMAL TO CHARACTER

DC is the basic byte-building subroutine used by MK and LB. It converts a decimal input between 0 and 255 to a byte, which is appended to alpha. This permits arbitrary strings of bytes to be assembled simply by specifying the corresponding decimal codes in sequence.

Example 1: Assemble a synthetic string for ASTO'ing. Key in "A", 40 XEQ DC, 12 XEQ DC, 41 XEQ DC, then append "=" and a space. You now have "A(μ) = " ready to be ASTO'ed for use in a program.

COMPLETE INSTRUCTIONS FOR DC

Enter a decimal number in X and XEQ DC. A character will be appended to the right of alpha (the last byte of the M register). Its decimal equivalent will be INT (x) mod 256. All characters previously in alpha will be preserved (except for the leftmost character of a string that was already 24 characters long). The previous contents of Y and Z are returned in X and Y. LastX contains INT(x) mod 256 + 256, which can be used if the same byte is needed again.

MORE EXAMPLES OF DC

Example 2: See the TN writeup for an example of how DC can be used in a program that creates and executes a synthetic "mini-program".

LINE BY LINE ANALYSIS OF DC

Lines 176-181 replace X by INT(x) mod 256 + 256. When this is converted to octal, the 256 part becomes 400_8.

Lines 183-194 move bits in the flag register to convert this octal number to hexadecimal (flags 08-15). Lines 195-197 save M and N in the stack, while O, P, and the new code are shifted one byte to the left. Lines 198-202 save O and P in the stack, while the former N and M are shifted to the left one byte. The new byte is simultaneously added at the right of the former M, now in N. Then P and O are restored and the former N and M are put back (lines 203-209). All four have been shifted one byte and the new byte has been attached.

CONTRIBUTORS HISTORY FOR DC

DC began as REBLD (see *PPC CALCULATOR JOURNAL*, V6N8P29), one of the original Black Box programs by Bill Wickes (3735). It was used as a subroutine for the original CODE program. The ROM version, written by Roger Hill (4940), uses an idea of Phillipe Roussel's (4367) to get the 24 character capability. The core of DC uses the built-in decimal-octal conversion routine in a way that was pioneered by Roger Hill. This very general and very powerful synthetic programming technique implements decimal-hexadecimal conversion by using octal conversion combined with flag operations to perform octal-hexadecimal conversion. This latter operation is merely bit shifting, and is very fast.

It also draws from versions by Charles Close (3878) and Carter Buck (4783). The core of DC uses the built-in decimal-octal conversion routine in a way that was pioneered by Roger Hill. This very general and very powerful synthetic programming technique implements decimal-hexadecimal conversion by using octal conversion combined with flag operations to perform octal-hexadecimal conversion. This latter operation is merely bit shifting, and is very fast.

Routine Listing For: DC

```
176♦LBL "DC"
177 INT
178 256
179 MOD
180 LASTX
181 +
182 OCT
183 X<> d
184 FS?C 11
185 SF 12
186 FS?C 10
187 SF 11
188 FS?C 09
189 SF 10
190 FS? 07
191 SF 09
192 FS? 06
193 SF 08
194 X<> d
195 X<> [
196 RCL \
197 "├*"
198 X<> ]
199 X<>Y
200 STO \
201 X<> ↑
202 "├*"
203 STO ↑
204 RDN
205 X<> ]
206 X<> \
207 STO [
208 RDN
209 END
```

TECHNICAL DETAILS

XROM: 10,11 — DC — SIZE: 000

Stack Usage:
- 0 T: new M
- 1 Z: new P
- 2 Y: Z
- 3 X: Y
- 4 L: X + 256

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

SHIFTED ONE CHARACTER TO THE LEFT.

Other Status Registers:
- 9 Q: NOT USED
- 10 ├: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED

Flag Usage: MANY USED BUT ALL RESTORED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: 1.5 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	LF
Bytes In RAM:	70
Registers To Copy:	59

Other Comments:

DC

STACK AND ALPHA REGISTER ANALYSIS FOR DC

T	Z	Y	X	L	P	O	N	M	L-#	INSTRUCTION
t	z	y	x	1	1234ABC	DEFGHIJ	KLMNOPQ	RSTUVWX	176	♦LBL "DC"
			INT(x)	x					177	INT
z	y	INT(x)	256						178	256
z	z	y	dec	256					179	MOD
z	y	dec	256						180	LASTX
z	z	y	256+dec						181	+
			400+oct	256+dec					182	OCT
			d						183	X<> d
									184	FS?C 11
									185	SF 12
									186	FS?C 10
									187	SF 11
									188	FS?C 09
									189	SF 10
									190	FS? 07
									191	SF 09
									192	FS? 06
									193	SF 08
			100+hex						194	X<> d
			M					⊼ # · · · · ⊠	195	X<> [
z	y	M	N						196	RCL \
					234ABCD	EFGHIJK	LMNOPQ	# · · · · ⊠ *	197	"⊢*"
			234ABCD			KLMNOPQ			198	X<>]
		EFGHIJK	M						199	X<>Y
							RSTUVWX		200	STO \
			234ABCD		RSTUVWX				201	X<> ↑
					STUVWXK	LMNOPQR	STUVWX#	· · · · ⊠ * *	202	"⊢*"
					234ABCD				203	STO ↑
234ABCD	z	y	EFGHIJK						204	RDN
			LMNOPQR			EFGHIJK			205	X<>]
			STUVWX#				LMNOPQR		206	X<> \
								STUVWX#	207	STO [
STUVWX#	234ABCD	z	y						208	RDN
STUVWX#	234ABCD	z	y	256+dec	234ABCD	EFGHIJK	LMNOPQR	STUVWX#	209	END

DF - DECIMAL TO FRACTION

This routine will convert a decimal to a fraction which approximates the decimal. The routine ends when the rounded value of the difference between a divided fraction and the decimal is zero. The accuracy of the approximation depends on a display setting which is a number between 0 and 9 which is stored in register R07. Setting flag F10 will display the resulting fraction in the alpha register. This routine makes it possible to use the calculator for decimal calculations (solving systems of equations or finding inverses of matrices or other problems) and yet return exact rational answers when all calculations are finished.

Example 1: Find a fraction which approximates the number pi = 3.141592654 to 2 decimal places.

Store the display setting in register R07. 2 STO 07. Set flag 10, SF 10, to see the display in fractional form. (You may also wish to clear flag F29). Key in pi and XEQ "DF". The display returns 22/7 where 22 is in the Y register and 7 is in the X register.

Example 2: Find a fraction which approximates the number pi = 3.141592654 to 4 decimal places.

Having just completed the previous example we need only change the display setting in R07. Key 4 STO 07 and then key in pi and XEQ "DF". The display returns 355/113 where 355 is in the Y register and 113 is in the X register.

Example 3: Find a fraction which approximates the natural log base e = 2.718281828 to 5 decimal places.

Store the display setting in R07. 5 STO 07. Key in e = 2.718281828 and XEQ "DF". The display returns 1264/465 where 1264 is in the Y register and 465 is in the X register.

DF

COMPLETE INSTRUCTIONS FOR DF

1) Store a display setting which is a number between 0 and 9 in register R07.

2) Flag 10 controls a display option. Setting flag F10 will result in displaying the answer in true fractional form in the alpha register. With F10 clear the alpha register is not used to display the final answer.

3) With the decimal value in the X register XEQ "DF". The resulting fraction is returned in the Y and X registers with the numerator in Y and the denominator in X. In addition, if F10 was set the fraction will be displayed in the alpha register. The fraction is in lowest terms. The denominator returned is always positive. In the case of negative decimals the numerator returned will be negative. The routine ends with a display setting of FIX n where n is the number in R07 if F10 was clear. If F10 was set then the routine ends in FIX 0.

MORE EXAMPLES OF DF

Example 4: Find a fraction which approximates pi = 3.141592654 to all 9 decimal places.

Key 9 STO 07. SF 10 to display the answer. Key in pi and XEQ "DF". The routine returns 104348/33215. Now press / and FIX 9 and see 3.141592654 in the X-register.

Example 5: Find the fraction represented by the decimal 0.263157895.

Leave 9 in R07 from the previous example but clear flag 10. CF 10. Key in .263157895 and XEQ "DF". See 19 in the X register when the routine ends. Then R↓ or X<>Y to see 5. The fraction is 5/19.

Example 6: Find the fraction represented by 0.

For this special example the number in R07 isn't used. Keep flag 10 clear. Key in 0 and XEQ "DF".

Y: 0
X: 1 0=0/1

Example 7: Key in any positive or negative whole number and follow the directions in Example 6.

FORMULAS USED IN DF

The technique used to find the approximating fraction in the DF routine depends on continued fractions. For additional information on this technique see the article in PPCJ V4N6P20 and V4N6P61. The iteration formulas the routine uses are:

x = original decimal to be approximated.

(1) $p_0 = x$

(2) $p_{i+1} = 1/(\ p_i - \mathrm{INT}(p_i)\ )$

(3) $D_{-1} = 0 \qquad D_0 = 1$

(4) $D_{i+1} = D_i * \mathrm{INT}(p_i) - D_{i-1}$

(5) $N_{i+1} = x * D_{i+1}$ rounded in FIX 0 mode

The fractions N_j/D_j approximate x.

Routine Listing For: DF

```
49♦LBL d
50♦LBL "DF"
51 STO 08
52 INT
53 .
54 STO 09
55 E
56 STO 10
57 RCL 08
58 R↑
59 X=Y?
60 GTO 08
61 ST- Y
62♦LBL 07
63 RDN
64 1/X
65 ENTER↑
66 INT
67 -
68 RCL 09
69 RCL 10
70 STO 09
71 LASTX
72 *
73 +
74 STO 10
75 RCL 08
76 *
77 FIX 0
78 RND
79 STO Z
80 RCL 10
81 /
82 RCL 08
83 -
84 FIX IND 07
85 RND
86 X≠0?
87 GTO 07
88 RCL Z
89♦LBL 08
90 RCL 10
91 SIGN
92 ST* 10
93 *
94 RCL 10
95 FC? 10
96 RTN
97 GTO 05
```

LINE BY LINE ANALYSIS OF DF

Lines 49-61 perform the intialization and check if a whole number has been input.

Lines 62-87 are the main loop in the program. Lines 62-73 calculate formula 4). Lines 75-78 calculate formula 5). The test at line 86 checks the rounded difference between the fraction approximation and the original decimal input. The loop is entered again until this difference is found to be zero.

Lines 88-97 end the routine by placing the correct sign on the numerator and deciding whether or not to display the result in the alpha register.

REFERENCES FOR DF

1. John Kennedy (918) "65 NOTES" Number Theory V4N6P17-20,P61

2. Charles G. Moore, "An Introduction To Continued Fractions," National Council of Teachers of Mathematics, 1964

CONTRIBUTORS HISTORY FOR DF

The DF routine and documentation were written by John Kennedy (918) based on an earlier HP-25 program related to continued fractions.

FURTHER ASSISTANCE ON DF

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

T E C H N I C A L   D E T A I L S

XROM: 20, 13 | DF | SIZE: 011

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: alpha registers used only when F10 is set
- 6 N:
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
- R00: not used
- R06: not used
- R07: accuracy factor
- R08: X
- R09: D_{i-1}
- R10: D_i
- R11: not used
- R12: not used

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: used to display fractional forms
- 25: used in VA routine

Display Mode:
FIX 9 DF may return FIX n, n=R07, or may return in FIX 0

Angular Mode:
not used

Unused Subroutine Levels:
4

Global Labels Called:

Direct	Secondary
VA called only if F10 set	none

Local Labels In This Routine:
d, 07, 08 (05, 06) if F10 set

Execution Time: depends on decimal and desired accuracy. Typical range 2-5 seconds

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: FR

Bytes In RAM: 68

Registers To Copy: 36

Other Comments:

DP - DECIMAL TO PROGRAM POINTER

DP converts a decimal input in X to a program pointer in RAM format suitable for a RAM STO b command. The decimal input is interpreted as the number of bytes from the last byte of the T register (i.e., from the bottom of program memory).

Example 1: DP is the inverse of PD. For instance do CAT 1 or XEQ GE to get into RAM. Then RCL b from the keyboard (see MK Example 1). Key ENTER ↑, XEQ PD, XEQ DP, X = Y?. You should see YES displayed. This indicates that DP exactly reversed the operation of PD on the RAM program pointer.

COMPLETE INSTRUCTIONS FOR DP

With a decimal number in X, XEQ DP to get a program pointer. This program pointer corresponds to the Xth byte from the bottom of RAM, e.g. x = 0 gives a pointer to the exponent byte of the T register. All of the stack is used except for Y, which is saved in Y.

APPLICATION PROGRAM 1 FOR DP

PA can be used from the keyboard to advance a program pointer by a specified number of bytes. However, PA cannot be used effectively in a program because it ends with XROM GE. To advance the program pointer in a running program, use the sequence

```
RCL b
XROM PD
k
-
XROM DP
STO b
```

NOTE: be sure to PACK before using this sequence. This eliminates the null preceding k.

For k = 0 the STO b will jump the program pointer right back to the XROM PD instruction. For k = - (2+m) the jump lands m bytes above the RCL b instruction. For k = 7 + m + n, where n is the number of digits in k, the jump lands m bytes below STO b; i.e., for m = 0 execution picks up with the instruction immediately following STO b.

For an interesting example, place a TONE IND 06 instruction immediately before the RCL b in the above sequence and use k = -3. Store a 9 in register 06 and run the program. You'll hear a TONE 9 then a BEEP every several seconds. The BEEP you hear is actually the second byte of the TONE IND 06 instruction. The STO b jumps right into the middle of the instruction.

LINE BY LINE ANALYSIS OF DP

Lines 67-70 separate the decimal input (number of bytes) into two parts: a number of 7-byte registers and a number of leftover bytes. The number of leftover bytes is multiplied by 16. The number of registers is decomposed into two base-256 components (lines 73-74). Lines 75-76 form the decimal sum to constitute the first byte of the pointer (16* leftover bytes + INT (register/256)). Lines 77-78 convert this to a byte in alpha, and line 79 attaches the second byte (registers mod 256). The code is recalled into X to complete the program.

TECHNICAL DETAILS

XROM: 10,53 — DP — SIZE: 000

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: Y
- 3 X: result
- 4 L: last byte + 256

Alpha Register Usage:
- 5 M: LOST
- 6 N: CLEARED
- 7 O: CLEARED
- 8 P: CLEARED

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 5

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Global Labels Called:

Direct	Secondary
QR DC	NONE

Local Labels In This Routine: NONE

Execution Time: 2.9 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	IF
Bytes In RAM:	28
Registers To Copy:	60

Other Comments:

DP

APPLICATION PROGRAM 2 FOR DP

DP constructs a progam pointer in RAM format. To construct a pointer in ROM format use the following routine.

```
LBL"DRP"
256
XROM QR
X<>Y
CLA
XROM DC
XROM DC
RCL M
RTN
```

This routine is the inverse of "RPD", Application Program 1 for PD If you want to use the pointer produced by "DRP" to jump into a ROM, you'll have to use Ab or Sb rather than just STO b in RAM. The reasons for this are discussed in the Ab write-up.

Routine Listing For: DP	
67♦LBL "DP"	75 X<> Z
68 7	76 +
69 XROM "QR"	77 CLA
70 X<>Y	78 XROM "DC"
71 16	79 XROM "DC"
72 ST* Z	80 RCL [
73 X↑2	81 RTN
74 XROM "QR"	

CONTRIBUTORS HISTORY FOR DP

DP started as relatively long routine (V7N7P18) by Keith Jarett (4360). Roger Hill (4940) completely rewrote DP as part of the PD / DP / CB / PA / RT / 2D / QR package to save many bytes. This made it possible to include other valuable routines in the ROM.

FURTHER ASSISTANCE ON DP

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

NOTES

DR - DELETE RECORD

This routine is called delete record and can be considered part of a file management system. DR applies to files consisting of fixed length records where each record is a block of consecutive data registers. DR is a special block move routine which deletes a given record from the file and moves the remaining records into the space occupied by the deleted record so that the data area is used as efficiently as possible. See also the related routine IR.

Example 1: The following list of registers shows an example file consisting of a simplified telephone directory. Use DR to delete the 4th record in this file.

Since this example file is used in the documentation of other ROM routines you may wish to record the file on a magnetic card for later use. This example file consists of a list of names and phone numbers. Only six records are in the file and each record consists of 6 consecutive registers with the following format:

1st register holds first name
2nd and 3rd registers hold the last name
4th register holds the telephone number
5th register holds the city name
6th register holds the state name

The records when printed would be the following:

Record #1:	Mary Adams 354-1662 Gary, IN
Record #2	Jane Hamilton 363-5648 Boston, MA
Record #3	Robert Jefferson 261-2347 Fresno, CA
Record #4	Mike Johnson 745-3254 Denver, CO
Record #5	James Masterson 565-2314 Toledo, OH
Record #6	Joe Robinson 756-4438 Peoria, IL

This sample file is stored in data registers R10-R45 where each record consists of 6 consecutive data registers.

R10: Mary	R28: Mike	R41: Robins
R11: Adams	R29: Johnso	R42: on
R12:	R30: n	R43: 756.4438
R13: 354.1662	R31: 745.3254	R44: Peoria
R14: Gary	R32: Denver	R45: IL
R15: IN	R33: CO	
R16: Jane	R34: James	
R17: Hamilt	R35: Master	
R18: on	R36: son	
R19: 363.5648	R37: 565.2314	
R20: Boston	R38: Toledo	
R21: MA	R39: OH	
R22: Robert	R40: Joe	
R23: Jeffer		
R24: son		
R25: 261.2347		
R26: Fresno		
R27: CA		

Like all the other file management routines, DR can expect to find the following information in registers R07, R08, and R09.

R07: starting register of entire file
R08: number of registers per record
R09: total number of records in the file

For the above sample file these numbers are:

R07: 10 = starting register
R08: 6 = number of registers per record
R09: 6 = total number of records

Having stored the data and the file information in the above registers, to delete record 4, Mike Johnson, simply key in 4 and XEQ "DR". The data registers when the DR routine ends contain the following.

R07: 10	R28: James
R08: 6	R29: Master
R09: 5	R30: son
R10: Mary	R31: 565.2314
R11: Adams	R32: Toledo
R12:	R33: OH
R13: 354.1662	R34: Joe
R14: Gary	R35: Robins
R15: IN	R36: on
R16: Jane	R37: 756.4438
R17: Hamilt	R38: Peoria
R18: on	R39: IL
R19: 363.5648	
R20: Boston	
R21: MA	
R22: Robert	
R23: Jeffer	
R24: son	
R25: 261.2347	
R26: Fresno	
R27: CA	

Note that DR simply moved the data following record 4 into the space previously occupied by record 4. Also DR updated the count of the total number of records in R09. Note that James Masterson is now the 4th record and Joe Robinson is now the 5th record.

COMPLETE INSTRUCTIONS FOR DR

1) A file in the 41C is to consist of a number of fixed length records where each record consists of a consecutive block of registers. Thus the entire file consists of one large block of consecutive registers. As with the other file management routines, DR assumes the following information is in registers R07, R08, and R09.

R07: starting register of the entire file
R08: number of consecutive registers per record
R09: total number of records in the file

2) To delete the kth record from the file key in k and XEQ "DR".

3) DR will move the records following the kth record into the registers occupied by the old kth record and will also subtract 1 from R09 to update the new number

of records. Note that this will cause a change in the numbering of the records following the kth record. The DR routine uses the block move routine BM.

MORE EXAMPLES OF DR

Example 2: The following matrix was used in Example 1 of the M1 routine. Matrices are assumed to be stored with each row occupying a consecutive block of registers. Thus the number of columns is the block size and the entire matrix is stored row by row as one string of consecutive registers. R07 holds the starting register of the matrix and R08 holds the number of columns. In this manner the storage of matrices corresponds to file storage and vice versa. As a result, the file management routines and the matrix manipulation routines can be used together. In the matrix routines M1 - M5 it is not necessary to store the number of rows in R09, but if either IR or DR is to be applied to a matrix, the user is advised to reserve R09 for the number of rows in the matrix. The following 6x5 matrix is assumed to be stored in registers R15-R44. Use DR to delete the 4th row from this matrix.

The original matrix is:

$$\begin{bmatrix} 21 & 35 & 55 & 74 & 83 \\ 11 & 93 & 56 & 36 & 29 \\ 65 & 78 & 32 & 27 & 75 \\ 53 & 94 & 46 & 62 & 97 \\ 54 & 39 & 61 & 67 & 82 \\ 23 & 45 & 77 & 15 & 25 \end{bmatrix}$$

and we show the correspondence between the data registers and the original matrix elements below. The element in the upper left-hand corner is assumed to be in row 1 and column 1. Store the matrix entries in the following registers.

R15: 21	R23: 36	R31: 94	R39: 82
R16: 35	R24: 29	R32: 46	R40: 23
R17: 55	R25: 65	R33: 62	R41: 45
R18: 74	R26: 78	R34: 97	R42: 77
R19: 83	R27: 32	R35: 54	R43: 15
R20: 11	R28: 27	R36: 39	R44: 25
R21: 93	R29: 75	R37: 61	
R22: 56	R30: 53	R38: 67	

Store the following data in R07, R08, and R09.

R07: 15 = starting register of matrix
R08: 5 = number of columns in the matrix
R09: 6 = number of rows in the matrix.

The rows of the matrix correspond to records in a file so to delete the 4th row, key in 4 and XEQ "DR". Rows 5 and 6 in the original matrix will move up in memory to occupy the space previously occupied by row 4. Note also that R09 now contains 5 for the new number of rows. The data registers now contain the following data.

R15: 21	R22: 56	R29: 75	R36: 45
R16: 35	R23: 36	R30: 54	R37: 77
R17: 55	R24: 29	R31: 39	R38: 15
R18: 74	R25: 65	R32: 61	R39: 25
R19: 83	R26: 78	R33: 67	
R20: 11	R27: 32	R34: 82	
R21: 93	R28: 27	R35: 23	

Routine Listing For: DR

```
 97♦LBL "DR"
 98 XEQ 03
 99 ST- Z
100 *
101 DSE 09
102 ""
103♦LBL "BM"
104 SIGN
105 RDN
106 X<Y?
107 GTO 04
108 LASTX
109 ST+ Z
110 +
111 -1
112 ST+ Z
113 ST+ Y
114 RDN
115♦LBL 04
116 R↑
117♦LBL 05
118 RCL IND Z
119 STO IND Z
120 RDN
121 ST+ Z
122 ST+ Y
123 DSE L
124 GTO 05
125 RTN

 78♦LBL 03
 79 RCL 07
 80 RCL 08
 81 RCL Z
 82 *
 83 +
 84 STO Y
 85 RCL 09
 86 R↑
 87 -
 88 RCL 08
 89 RTN
```

LINE BY LINE ANALYSIS OF DR

DR uses the BM routine and updates the count in R09. The input required of BM is given in the following stack configuration where s=starting register of the file, c=number of registers per record, n=number of records in the file. I= user input to DR.

Z: 1st register = s + c*I
Y: destination of 1st reg. = s + c*(I-1)
X: number of registers = c*(n-I)

Lines 97-100 set up these values in the stack before control drops through to BM. LBL 03 is a special subroutine used by both DR and IR to help set up the stack values.

CONTRIBUTORS HISTORY FOR DR

The DR routine and documentation were written by John Kennedy (918).

FINAL REMARKS FOR DR

DR is barely a start for a file management system on the calculator.

FURTHER ASSISTANCE ON DR

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

NOTES

TECHNICAL DETAILS

XROM: 20, 38	DR	SIZE: depends on file size

Stack Usage: 0 T:used 1 Z:used 2 Y:used 3 X:used 4 L:used	**Flag Usage:** 04: not used 05: not used 06: not used 07: not used 08: not used 09: not used 10: not used 25: not used
Alpha Register Usage: 5 M:not used 6 N:not used 7 O:not used 8 P:not used	
Other Status Registers: 9 Q:not used 10 ⊢:not used 11 a:not used 12 b:not used 13 c: not used 14 d: not used 15 e: not used	**Display Mode:** not used not used **Angular Mode:** not used **Unused Subroutine Levels:** 5
ΣREG: not used **Data Registers:** R00: The data registers used are those required by the storage of the file R06: R07: R08: R09: R10: R11: R12:	**Global Labels Called:** Direct: none Secondary: none **Local Labels In This Routine:** 03
Execution Time: depends on file size and configuration as well as the number of records. See BM	
Peripherals Required: none	
Interruptible? yes Execute Anytime? no Program File: M2 Bytes In RAM: 71 Registers To Copy: 61	**Other Comments:** No special SIZE requirement is necessary provided the data block already exists

APPENDIX E ROM PROJECT EXPENSE SUMMARY

The PPC ROM project is a special project of PPC. This means that it is not a 100% PPC project, but rather a member proposed and managed project that is PPC sanctioned and supported. See the *PPC MEMBER HANDBOOK*, 2nd. Edition, page 71 for additional comments on club projects.

This project was first proposed to the membership by me in the August 1979 issue of the *PPC JOURNAL*. Formal cost estimates and ordering estimates were published in the September 1980 member letter. Because of the large number of unknowns involved, it was decided to estimate the number of members interested in participating at 500, and project all costs on this number. We exceeded this number by a factor of 5.

2,500 orders were accepted at the following rates depending on shipment costs to the members region.

To Europe....................	$U.S. 99.21
To Asia......................	$U.S. 100.90
To South & Central America...	$U.S. 97.52

All monies were received by me and deposited into non-interest checking account No. 911-0-62852 at the Bank of America, Bristol-McFadden branch, Santa Ana, California. When all orders were in, the account was closed and all monies transferred to A/N 911-01625. This was necessary to prevent acceptance of funds that were wire transferred by people hearing of the Project after the close date. All monies were spent on materials and services necessary to produce the ROM and its documentation. No member received any payment for work done on the ROM. Every ROM ordered, 5,000 pieces, was serialized according to the 1 thru 2500 orders. Each order received two ROM's engraved with NNNNA and NNNNB. The list of orders (members) is included herein as Appendix F, page 135 (135, 143, 165, 169, 173). The "books" were audited by Dick Nobel (9) prior to shipment. The following preliminary summary outlines all monies involved in this project. I personally managed all funds. The approximate balance remaining will be used to reprint the Pocket Guide and/or Addendum discussed in the Foreword.

Richard J. Nelson (1)

		Expenses	Received
Hewlett-Packard			
Mask Charge........................	$ 17,000.00		
SDS System Rental..................	570.00		
ROM Production (5,000 @ $22 ea.)...	110,000.00		
Total	$127,570.00	$127,570.00	
Printing			
Art Work...........................	$ 750.00		
Pocket Guide.......................	3,498.00		
Plastic Hex Card...................	2,500.00		
Manual.............................	54,000.00		
Labels.............................	493.68		
Total	$ 61,241.68	$ 61.241.68	
Shipping - 12 lb., 1 ft.^{3} U.S......	$ 14,500.00		
NON - U.S......	4,200.00		
Total	$ 18,700.00	$ 18,700.00	
Preparation			
Packaging..........................	$ 1,750.00		
Telephone & Supplies...............	685.00		
Waxer & Wax........................	609.39		
Engraving..........................	1,226.47		
Clerical Service (Typing)..........	2,830.00		
Supplies...........................	325.00		
Total	$ 7,425.86	$ 7,425.86	
		$214,937.54	$238,427.31
		Balance on hand	$ 23,489.77

NOTES: All amounts are approximate. Items such as supplies, telephone, and shipping do not include final costs. The balance on hand for the Addendum and reprinting of the Pocket Guide should be at least $15,000. The Addendum will contain the final expense report.

Richard J. Nelson (1)

END

DS - DISPLAY SET

DS provides a capability similar to the HP-67/97 DSP function, which sets the number of decimal places to be displayed without changing the display mode type (FIX, ENG, or SCI).

Example 1: Set FIX 2, then do 5 XEQ DS. The display mode should change to FIX 5. Set ENG 2, then 3 XEQ DS to get ENG 3. Set SCI 8, then 7 XEQ DS to get SCI 7.

COMPLETE INSTRUCTIONS FOR DS

Put a number in x whose integer part is between 0 and 9 inclusive. XEQ DS to select the designated number of decimal places to be displayed. The display mode type (FIX, ENG, OR SCI) is not changed. This operation is analogous to the HP-67/97 function DSP. In fact, DS can be regarded as DSP IND X (followed by RDN). DS saves Y, Z, and T in X, Y, and Z, and saves X in L. The alpha register is cleared, and the new flag register contents are copied into T.

LINE BY LINE ANALYSIS OF DS

Lines 42-46 copy X into L and copy the flag register into X and 0 to allow the display mode type to be retained. The number of digits is copied into flags 36-39, the first nybble of the fifth byte of the flag register.

This new flag register is then copied into M and shifted left 5 bytes. At this point the first five bytes of the new flag register are in N and the last two bytes of the old flag register are in O. Lines 51-58 assemble these two pieces, store the result in the flag register, and clear out the alpha register.

See Page 137 For Stack and Alpha Analysis

CONTRIBUTORS HISTORY FOR DS

DS was written by Keith Jarett (4360).

Routine Listing For: DS

```
42♦LBL "DS"      51 X<> ]
43 SIGN          52 STO [
44 RDN           53 "⊢**"
45 RCL d         54 X<> \
46 STO ]         55 STO d
47 SCI IND L     56 RDN
48 X<> d         57 CLA
49 STO [         58 RTN
50 "⊢*****"
```

FURTHER ASSISTANCE ON DS

Call Keith Jarett (4360) at (213) 374-2583.
Call Keith Kendall (5425) at (801) 967-8080.

TECHNICAL DETAILS

XROM: 10,29 — DS — SIZE: 000

Stack Usage:
- 0 T: new d
- 1 Z: T
- 2 Y: Z
- 3 X: Y
- 4 L: X

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R06:
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Flag Usage: NONE USED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10:
- 25:

Display Mode: NUMBER OF DIGITS DISPLAYED SET AS DESIGNATED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: .6 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	VM
Bytes In RAM:	40
Registers To Copy:	60

Other Comments:

DS

APPENDIX A CONTINUED FROM PAGE 61.

```
01♦LBL "IGT"
02 CLRG
03 STO 02
04 X<>Y
05 STO 03
06 8
07 +
08 XROM "VS"
09 FC?C 25
10 PROMPT
11 RCL 03
12 STO 04
13 0

14♦LBL 00
15 RCL Y
16 +
17 E1
18 /
19 DSE Y
20 GTO 00
21 STO 01
22 RCL 02
23 5
24 /
25 ENTER↑
26 INT
27 X=Y?
28 GTO 01
29 1
30 +

31♦LBL 01
32 STO 00
33 XEQ "PARTS"
34 RCL 00
35 STO 05
36 RCL 09
37 E1
38 ST/ 04
39 ST/ 04
40 ST/ 05
41 DSE X
42 *
43 RCL 03
44 2
45 -
46 ST+ Y
47 3
48 -
49 X<=0?
50 ST- Y
51 RDN
52 STO 06
53 RCL 05
54 +
55 ST+ 04
56 INT
57 RCL 03
58 RCL 00
59 *
60 +
61 1
62 +
63 STO 07
64 RCL 09
65 1
66 -
67 5
68 /
69 INT
70 2
71 ST* Y
72 +
73 +
74 XROM "VS"
75 FC?C 25
76 PROMPT
77 RCL 06
78 -1
79 STO IND Y
80 RCL 07
81 XEQ "IRX"
82 RCL 06
83 RCL 02
84 E3
85 /
86 1
87 ST+ Z
88 +
89 .5
90 ST+ Z

91♦LBL 05
92 CLX
93 5
94 STO 05
95 CLX

96♦LBL 06
97 E2
98 *
99 RCL Y
100 INT
101 +
102 DSE 05
103 GTO 07
104 STO IND Z
105 ISG Z
106 ISG Y
107 GTO 05
108 GTO 10

109♦LBL 07
110 ISG Y
111 GTO 06
112 STO IND Z

113♦LBL 10
114 "READY?"
115 PROMPT
116 XEQ "SHOW"
117 GTO "GHT"
118 END

01♦LBL "GHT"
02 XEQ "LRR"
03 XEQ "PARTS"
04 7.5
05 ENTER↑
06 5
07 X<> 06
08 0

09♦LBL 02
10 E2
11 *
12 RCL IND Y
13 +
14 DSE 06
15 GTO 03
16 STO IND Z
17 X<> 06
18 ISG Z

19♦LBL 03
20 ISG Y
21 GTO 02
22 STO IND Z
23 RCL 03
24 2
25 -
26 STO 05
27 RCL 01
28 E1
29 *
30 INT
31 STO 06
32 RCL 03
33 STO [
34 10↑X
35 RCL 01
36 *
37 DSE [

38♦LBL 04
39 E1
40 ST* 06
41 /
42 ENTER↑
43 FRC
44 E1
45 *
46 INT
47 ST+ 06
48 RDN
49 DSE [
50 GTO 04

51♦LBL 05
52 5
53 RCL 05
54 X>Y?
55 GTO 06
56 RCL 07
57 E2
58 /
59 STO 07
60 GTO 07

61♦LBL 06
62 RCL 08
63 E2
64 /
65 STO 08

66♦LBL 07
67 FRC
68 E2
69 *
70 INT
71 X=0?
72 GTO 09
73 XEQ 70
74 RCL 06
75 RCL 05
76 2
77 +
78 FS?C 01
79 GTO 08
80 STO 12
81 10↑X
82 /
83 STO 10
84 XEQ 75
85 XEQ "GHT"
86 XROM "UD"
87 GTO 09

88♦LBL 08
89 10↑X
90 /
91 E1
92 *
93 INT
94 RCL 06
95 E1
96 /
97 FRC
98 E1
99 *
100 X<>Y
101 XEQ "MOVE"

102♦LBL 09
103 RCL 06
104 E1
105 /
106 INT
107 STO 06
108 DSE 05
109 GTO 05
110 RCL 01
111 RCL 03
112 1
113 -
114 10↑X
115 *
116 FRC
117 E1
118 *
119 RCL 01
120 E1
121 *
122 INT
123 XEQ "MOVE"
124 RCL 03
125 2
126 -
127 STO 06
128 E3
129 /
130 1
131 ST+ 06
132 +
133 STO 05
134 RCL 06
135 10↑X
136 RCL 01
137 *
138 STO 06

139♦LBL 10
140 5
141 RCL 05
142 INT
143 X>Y?
144 GTO 11
145 RCL 07
146 RCL 07
147 E2
148 *
149 STO 07
150 GTO 12

151♦LBL 11
152 RCL 08
153 RCL 08
154 E2
155 *
156 STO 08

157♦LBL 12
158 INT
159 X<>Y
160 INT
161 E2
162 *
163 -
164 X=0?
165 GTO 15
166 XEQ 70
167 RCL 06
168 E1
169 /
170 STO 06
171 FRC
172 E1
173 *
174 FRC
175 LASTX
176 INT
177 RCL 01
178 E1
179 ST* Z
180 *
181 INT
182 +
183 +
184 FS?C 01
185 GTO 13
186 E2
187 /
188 STO 10
189 RCL 05
190 INT
191 2
192 +
193 STO 12
194 XEQ 75
195 XEQ "GHT"
196 XROM "UD"
197 GTO 14

198♦LBL 13
199 INT
200 LASTX
201 FRC
202 RCL 05
203 INT
204 1
205 -
206 10↑X
207 *
208 FRC
209 E1
210 ST/ Z
211 *
212 X<>Y
213 INT
214 XEQ "MOVE"

215♦LBL 14
216 ISG 05
217 GTO 10

218♦LBL 15
219 XEQ "SRR"
220 RTN

221♦LBL 70
222 CF 01
223 1
224 X<>Y
225 X=Y?
226 SF 01
227 FC? 01
228 STO 11
229 RTN

230♦LBL 75
231 RCL 04
232 STO 13
233 9
234 ST- 13
235 XROM "HD"
236 END

01♦LBL "PARTS"
02 RCL 02
03 STO 05
04 1
05 ST- 05
06 RCL 03
07 2
08 -
09 E3
10 /
11 +
12 8.008
13 +
14 STO 06
15 STO 07
16 0

17♦LBL 00
18 STO IND 07
19 ISG 07
20 GTO 00

21♦LBL 01
22 1
23 RCL 06
24 STO 07

25♦LBL 02
26 X<>Y
27 ST+ IND 07
28 ST- 05
29 RCL IND 07
30 RCL 05
31 X>0?
32 GTO 03
33 X=0?
34 RTN
35 ST+ IND 07
36 RTN

37♦LBL 03
38 ISG 07
39 GTO 02
40 GTO 01
41 END

01♦LBL "MOVE"
02 E2
03 STO 10
04 RDN
05 XEQ 20
06 ST* Y
07 ST* Z
08 ST- Z
09 RDN
10 RCL 04
11 INT
12 ST+ Y
13 1.5
14 +
15 ST+ Z
16 RDN
17 STO [
18 X<>Y
19 STO \
20 0
21 STO 09

22♦LBL 01
23 RCL IND [
24 X=0?
25 GTO 03
26 XEQ 25
27 10↑X
28 STO Z
29 /
30 ENTER↑
31 INT
32 X<> 09
33 CF 00
34 X=0?
35 SF 00
36 STO T
37 RDN
38 FRC
39 *
40 FS?C 00
41 GTO 02
42 RCL 10
43 *
44 +

45♦LBL 02
46 STO IND [

47♦LBL 03
48 DSE [
49 DSE ]
50 GTO 01
51 RCL 09
52 STO [
53 XEQ 20
54 CF 00

55♦LBL 05
56 E8
57 RCL IND
58 X<Y?
59 SF 00
60 RCL [
61 STO Z
62 RDN
63 FS? 00
64 GTO 06
65 RCL X
66 RCL 10
67 MOD
68 STO [
69 RDN
70 RCL 10
71 /
72 INT

73♦LBL 06
74 X=0?
75 GTO 07
76 XEQ 25
77 2
78 +
79 10↑X
80 ST* Z
81 RDN

82♦LBL 07
83 +
84 STO IND \
85 FS?C 00
86 GTO 10
87 ISG \
88 DSE ]
89 GTO 05

90♦LBL 10
91 GTO "SHOW"

92♦LBL 20
93 RCL 04
94 FRC
95 E1
96 *
97 INT
98 STO ]
99 RTN

100♦LBL 25
101 ENTER↑
102 LOG
103 2
104 /
105 INT
106 2
107 *
108 END

01♦LBL "SHOW"
02 FIX 0
03 CF 29
04 RCL 04
05 RCL IND X
06 1
07 +
08 STO IND Y
09 X=0?
10 GTO 10
11 "---MOVE "
12 ARCL X
13 "├---"
14 BEEP
15 ADV
16 ADV
17 XROM "VA"

18♦LBL 10
19 RCL 04
20 INT
21 RCL 04
22 FRC
23 E1
24 *
25 STO 09
26 INT
27 -
28 STO 10
29 RCL 09
30 FRC
31 E2
32 /
33 1
34 +
35 STO 09

36♦LBL 00
37 ADV
38 "PEG "
39 ARCL 09
40 XROM "VA"
41 TONE 7
42 TONE 7
43 RCL 10
44 RCL 10
45 RCL 04
46 FRC
47 E1
48 *
49 INT
50 ST+ 10
51 ST+ Y
52 ST+ Z
53 +
54 E3
55 /
56 +
57 1
58 +
59 STO 11
60 RCL IND X
61 X≠0?
62 GTO 01
63 TONE 4
64 TONE 4
65 "  **EMPTY**"
66 XROM "VA"
67 GTO 04

68♦LBL 01
69 RCL IND 11
70 X=0?
71 GTO 04
72 "     "
73 ENTER↑

74♦LBL 02
75 RDN
76 E2
77 /
78 ENTER↑
79 INT
80 X≠0?
81 GTO 02
82 RDN

83♦LBL 03
84 "├."
85 E2
86 *
87 ENTER↑
88 INT
89 E1
90 X>Y?
91 "├ "
92 ARCL Y
93 RDN
94 -
95 X≠0?
96 GTO 03
97 TONE 4
98 TONE 4
99 XROM "VA"
100 ISG 11
101 GTO 01

102♦LBL 04
103 ISG 09
104 GTO 00
105 END
```

LBL'IGT END	187 BYTES
LBL'GHT END	359 BYTES
LBL'PARTS END	68 BYTES
LBL'MOVE END	176 BYTES
LBL'SHOW END	190 BYTES

END

DT - DISPLAY TEST

DT is a brief, stand-alone routine that allows the user to verify that all segments of the display are operational. First, 12 commas appear in the display for the duration of one PSE cycle; then all annunciators and segments except for comma tails appear for as long as desired. This display so far holds the record for most segments lit in a 41 C display at once. DT is not designed to be used in a running program, except perhaps as an interesting way to end one.

COMPLETE INSTRUCTIONS FOR DT

1. XEQ DT - You will see the commas appear, then the complete display. Study the display for as long as necessary to make sure that all of the segments except the comma tails are visible.

2. Switch out of PRGM mode, and press R/S to restore previous machine status. NOTE: If you have any other function assigned to the R/S key, switch out of USER mode before pressing R/S. If you omit this step accidentally, use RF or X<>d to recover normal flag status.

3. To study the commas XEQ DT and push R/S to interrupt the program as soon as they appear. Push R/S to continue and proceed as above.

LINE BY LINE ANALYSIS OF DT

Lines 78 and 79 create a flag register code with all annunciators lit. Lines 80-84 display the 12 commas. Lines 85-90 create the alternating starburst/colon display, while lines 91-93 place the flag register code in d. Lines 94-97 restore the flag register and stack (except T and L).

CONTRIBUTORS HISTORY FOR DT

DT was conceived by Valentin Albillo (4747). His program appeared in the *PPC CALCULATOR JOURNAL*, V7N5P18. This DT program is a slightly modified version of the one that appeared in *SYNTHETIC PROGRAMMING* by Bill Wickes (3735).

Routine Listing For: **DT**

```
77♦LBL "DT"          87 ASTO L
 78 "♦♦8♦!           88 ARCL L
      "              89 ARCL L
79 RCL [             90 ARCL L
80 ",,,,,,"          91 X<> d
81 ASTO L            92 AVIEW
82 ARCL L            93 STOP
83 AON               94 X<> d
84 PSE               95 RDN
85 AOFF              96 CLD
 86 ":::"            97 RTN
```

FURTHER ASSISTANCE ON DT

Call William C. Wickes (3735) at (503) 754-0117.
Call Tom Cadwallader (3502) at (406) 727-6869.

TECHNICAL DETAILS

XROM: 10,17 | **DT** | SIZE: 000

Stack Usage:
- 0 T: temporary d
- 1 Z: UNCHANGED
- 2 Y: UNCHANGED
- 3 X: UNCHANGED
- 4 L: "✕:✕:✕:"

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

12 starbursts and 12 colons

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED

R00:

R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: MANY USED, BUT ALL RESTORED

04:
05:
06:
07:
08:
09:
10:

25:

Display Mode: SCI 0, BUT ORIGINAL MODE RESTORED BY SWITCHING TO RUN MODE AND PRESSING R/S.

Angular Mode: GRAD, BUT ORIGINAL MODE RESTORED BY SWITCHING TO RUN MODE AND PRESSING R/S.

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine:

NONE

Execution Time: 3.5 seconds.

Peripherals Required: NONE

Interruptible?	NO
Execute Anytime?	YES
Program File:	ML
Bytes In RAM:	54
Registers To Copy:	64

Other Comments:

MEM	QUAN	ORD
1	1	1
9	3	565
		2101
		2102
17	3	1761
		2321
		2322
22	1	27
24	1	1897
30	1	1293
60	1	1655
62	1	761
66	1	1553
74	1	2310
76	1	244
80	1	248
91	1	757
93	1	548
103	1	1047
110	1	425
111	1	37
118	1	107
136	1	1026
137	1	221
139	1	553
143	1	870
150	1	158
152	1	31
157	1	750
170	1	1657
181	1	1958
190	1	187
192	1	1569
217	1	944
226	1	571
228	1	2022
233	3	331
		2473
		2474
236	1	1524
239	1	1124
260	1	844
265	3	887
		1011
		1845
268	1	861
277	1	3
286	1	2429
289	2	448
		449
303	1	94
308	1	655
311	2	1572
		1718
327	3	1168
		1169
		1170
337	1	83
339	2	714
		715
348	1	378
352	1	1432
353	1	430
370	1	1258
404	1	936
418	1	196
423	1	779
438	1	309
440	3	160
		2191

MEM	QUAN	ORD
		2192
442	1	2168
453	1	66
458	1	202
481	4	559
		560
		561
		762
499	1	1245
503	1	1199
507	1	2206
533	1	706
534	2	64
		65
546	3	1684
		1685
		1686
552	2	719
		2134
554	1	1193
572	1	223
583	1	179
591	1	260
598	1	1185
602	1	724
611	1	838
618	1	1250
620	1	840
624	1	2049
628	1	494
629	2	1234
		1235
638	1	1924
643	1	74
644	1	1654
649	1	658
662	1	89
690	1	338
699	1	1320
700	1	530
701	1	2082
716	1	1301
719	1	595
731	1	1186
738	1	1903
739	1	1354
744	1	1364
746	1	1307
747	2	1012
		1013
756	1	195
760	1	456
766	2	141
		142
780	1	57
784	1	650
786	1	599
787	1	1044
799	1	597
803	1	1321
807	1	394
810	1	491
812	1	2464
818	2	70
		71
822	1	965
823	1	1766
830	1	98
834	1	857
847	1	147

MEM	QUAN	ORD
848	2	577
		578
851	1	1550
865	1	2185
871	1	96
877	1	1030
892	2	582
		583
899	1	602
900	1	784
901	3	745
		746
		898
912	1	435
916	1	2084
918	1	411
926	1	569
930	1	2466
934	1	2350
945	1	6
950	1	1810
960	1	1813
974	1	188
985	1	2017
997	1	2419
998	1	985
1001	1	191
1002	1	44
1004	1	2323
1016	1	268
1025	1	291
1047	2	624
		625
1049	1	310
1053	1	999
1063	1	8
1065	1	782
1067	1	1451
1086	1	245
1097	1	662
1115	1	2260
1119	1	2385
1127	1	199
1135	1	170
1136	1	843
1142	1	123
1145	1	43
1150	1	326
1158	1	146
1163	1	1937
1164	2	166
		167
1166	1	205
1167	1	930
1183	1	2138
1185	2	886
		1876
1198	1	1558
1200	1	490
1201	1	742
1207	1	334
1211	1	1015
1221	2	751
		752
1224	1	558
1232	1	1296
1236	1	180
1237	1	520
1241	2	1389
		1390

MEM	QUAN	ORD
1254	1	619
1259	1	581
1262	1	2286
1288	1	1892
1294	10	1794
		1795
		1796
		1797
		1798
		1799
		1800
		1801
		1802
		1803
1302	2	1181
		1182
1306	1	2211
1317	1	1440
1318	1	469
1319	1	1187
1329	1	1200
1331	1	1551
1338	1	340
1342	1	148
1348	1	151
1350	1	1700
1355	2	679
		680
1363	2	375
		376
1381	1	731
1390	1	1462
1402	2	871
		872
1404	1	2472
1414	1	1743
1449	1	793
1456	1	888
1480	1	2001
1481	1	963
1488	1	735
1500	1	251
1509	1	1216
1510	1	2169
1514	1	130
1520	1	131
1526	1	220
1533	1	1947
1540	2	426
		427
1542	1	1430
1552	1	689
1556	1	1120
1560	1	763
1571	1	1895
1592	1	1372
1593	1	1155
1594	1	1468
1597	2	1021
		1022
1605	1	867
1614	1	1305
1615	1	1264
1623	1	1695
1642	1	924
1651	1	1300
1652	4	517
		518
		1877
		1878

MEM	QUAN	ORD
1658	1	1371
1664	1	1009
1672	1	1355
1688	1	404
1702	1	660
1707	1	940
1709	1	648
1714	1	1050
1716	2	502
		1322
1718	2	1178
		1831
1719	2	631
		632
1725	1	570
1733	1	1421
1740	1	2349
1742	1	2339
1750	1	977
1755	1	419
1757	1	1114
1761	1	1504
1769	1	1280
1775	2	287
		288
1787	1	1221
1788	1	1530
1789	1	733
1793	1	126
1801	1	102
1803	2	1652
		1653
1804	1	483
1806	1	145
1816	1	13
1820	1	78
1822	1	1278
1823	1	1616
1824	1	1469
1826	1	900
1827	1	26
1838	1	197
1840	1	2236
1846	2	284
		285
1850	1	670
1851	1	1593
1855	1	186
1862	1	811
1869	1	642
1891	1	1601
1905	1	2426
1908	1	1400
1909	1	56
1913	1	1140
1918	1	162
1920	1	615
1927	1	1294
1939	1	498
1951	2	758
		759
1957	3	958
		959
		960
1959	3	108
		2253
		2254
1966	1	1150
1981	1	52
1984	1	321

MEM	QUAN	ORD
2001	1	2317
2002	1	848
2020	1	2420
2040	1	5
2046	1	328
2053	1	1334
2056	1	652
2057	1	439
2060	2	1501
		1502
2072	1	2262
2105	3	477
		478
		479
2114	1	585
2137	1	1072
2142	1	755
2147	1	1138
2153	1	726
2156	1	218
2157	1	1174
2169	1	1014
2175	1	1503
2178	1	663
2180	1	7
2193	1	1236
2210	1	617
2223	1	2044
2232	1	1774
2234	1	395
2249	1	1623
2253	1	100
2255	1	269
2256	1	1361
2265	1	659
2279	1	981
2280	1	2319
2284	2	300
		301
2285	1	904
2289	1	297
2305	1	398
2324	1	698
2325	1	290
2331	1	781
2339	1	152
2340	1	2018
2341	1	418
2357	1	1330
2388	1	1591
2392	2	836
		1263
2400	1	168
2405	1	618
2406	3	1271
		1272
		1273
2411	1	1999
2430	1	2093
2433	1	1955
2447	1	374
2454	1	86
2455	1	1843
2458	1	1936
2472	1	2229
2476	1	116
2478	1	2428
2482	2	1104
		1105
2485	5	1532

MEM	QUAN	ORD
		1533
		1534
		1535
		2488
2487	1	72
2490	1	1965
2511	1	1984
2517	1	1809
2520	1	1660
2523	1	1299
2525	1	922
2528	1	487
2529	1	531
2544	3	1386
		1387
		1388
2546	2	805
		806
2551	2	1677
		1678
2557	1	2347
2565	1	1456
2572	1	514
2578	1	1383
2585	3	1067
		1068
		1069
2593	1	800
2595	1	951
2596	1	241
2602	1	255
2604	1	2078
2607	1	801
2623	1	1071
2628	1	2437
2631	1	1963
2636	1	87
2647	1	1510
2648	1	134
2650	1	996
2659	1	1991
2666	1	379
2674	2	211
		213
2678	1	1327
2680	1	484
2681	2	579
		1526
2697	1	243
2719	1	238
2724	1	1752
2729	1	1108
2739	1	884
2748	1	802
2749	1	869
2750	1	1588
2754	1	1362
2755	2	302
		2056
2759	1	36
2761	1	69
2763	1	20
2764	1	164
2782	1	127
2785	1	214
2786	1	1265
2787	1	1688
2788	1	1900
2789	1	103
2791	2	637
		638

APPENDIX F CONTINUED ON PAGE 143

E? - .END. FINDER

E? locates the register currently containing the permanent .END. and returns its absolute address to the X-register as a decimal number.

Example 1: E? can be used together with PPC ROM routine CX to clear all data registers and all user program registers other than a user-specified block at the bottom of memory, while preserving the contents of the status and key assignment registers:

1.	XEQ E?	322*	Absolute address (decimal) of the register containing the .END.
2.	1+**	323	Absolute address of the next higher register.
3.	XEQ CX		CX lowers the "curtain" so that R00 is now defined to be the register just above that containing the .END.
4.	CLRG		All registers higher than that containing the .END. are cleared. Resize as desired; at most 4 bytes of program material will be left in the .END. register and these may be cleared manually.

* Representative number only.
**To save the bottom n-1 registers, substitute n+ for step 2.

COMPLETE INSTRUCTIONS FOR E?

E? does not require an input. It may be executed either manually or as a program routine. In either case, the routine ends with the absolute address of the register containing the permanent .END. returned to the X-register and expresses as a decimal number. The permanent .END. is contained in the block of registers available for user program memory--that is, from register $0C0_{16} = 192_{10}$ to the top of memory, which ranges from $0FF_{16} = 255_{10}$ (no memory modules) to $1FF_{16} = 511_{10}$ (four memory modules). For example, in a case where data registers and user programs occupy all available space in user memory and no key assignments have been made, the permanent .END. will be located in register $0C0_{16}$; execution of E? will return the decimal number 192 to the X-register.

The original contents of register X and Y are preserved by E? in Y and Z, respectively.

MORE EXAMPLES OF E?

Additional examples of the use of E? may be found in the PPC ROM routines CK and PS, which use E? as a subroutine.

APPLICATION PROGRAM 1 FOR E?

On occasion, it may be helpful to determine the number of registers occupied by user program memory or the number of registers available for additional program material, including any registers currently occupied by key assignments. "PRU" is a short program that uses the PPC ROM routines E? and C? to provide both in a friendly manner.

E? Support Program "PRU"

(Program Register Usage)

```
01 LBL "PRU"        10 "PRGM-"
02 XROM C?          11 ARCL Z
03 XROM E?          12 AON
04   -              13 PSE
05 LASTX            14 "AVAIL-"
06 192              15 ARCL Y
07   -              16 PSE
08 RCL d            17 STO d
09 FIX 0            18 END
```

When executed, PRU will pause with the display "PRGM - lmn", where lmn is the number of registers occupied by user programs (including the register containing the .END.). The program may be halted, with this display locked in, by pressing R/S during the pause. When restarted (after the pause or by pressing R/S a second time), the program will pause again with the display "AVAIL - pqr", where pqr is the number of registers available for additional programs, including registers below the .END. currently used as key assignment registers or containing other data. This display may also be locked in with R/S. As an alternative to the use of PPC ROM routine A?, a comparison of pqr with the number of registers shown by the calculator as available (after GTO.., PRGM) will reveal how many potential program registers are currently occupied, in whole or part, by data below the .END. register. All of those registers can be cleared through use of the PPC ROM routine CK.

Routine Listing For: E?

```
195♦LBL "E?"      201 X↑2
196 RCL c         202 *
197 XROM "2D"     203 RCL [
198 16            204 +
199 MOD           205 CLA
200 LASTX         206 END
```

LINE BY LINE ANALYSIS OF E?

The absolute address of the memory register containing the permanent .END. is maintained by the HP-41C/V as the last 1 1/2 bytes, or 3 hexadecimal nybbles pqr, of status register c. Lines 196-197 make use of the PPC ROM routine 2D to decode each of the last two bytes xpqr of register c, returning the decimal equivalent of xp to the X-register, and that of qr to register M. Lines 198-199, by taking xp modulo 16, produce the decimal equivalent of the p nybble alone. Lines 200-204 complete the decoding process by multiplying the p nybble by 256 and adding to the product the previously decoded qr byte stored in register M.

REFERENCES FOR E?

PPC Calculator Journal, V8N2P37. Earlier versions can be found in V7N6P10b and V7N7P16a.

CONTRIBUTORS HISTORY FOR E?

E? was written by Roger Hill (4940). Earlier versions were written by Keith Jarett (4360) and Valentin Albillo (4747).

TECHNICAL DETAILS		
XROM: 10,62	E?	SIZE: 000

Stack Usage:	**Flag Usage:** NONE USED
0 T: Y	04:
1 Z: Y	05:
2 Y: X	06:
3 X: result	07:
4 L: E? MOD 256	08:
Alpha Register Usage:	09:
5 M:	10:
6 N: ALL CLEARED	
7 O:	
8 P:	25:
Other Status Registers:	**Display Mode:** UNCHANGED
9 Q:	
10 ⊢:	
11 a: NONE USED	**Angular Mode:** UNCHANGED
12 b:	
13 c:	
14 d:	**Unused Subroutine Levels:**
15 e:	5
ΣREG: UNCHANGED	**Global Labels Called:**
Data Registers: NONE USED	**Direct** / **Secondary**
R00:	2D / NONE
R06:	
R07:	
R08:	
R09:	
R10:	
R11:	**Local Labels In This Routine:**
R12:	NONE

Execution Time: 1.9 seconds.

Peripherals Required: NONE

Interruptible? YES	**Other Comments:**
Execute Anytime? YES	
Program File: IF	
Bytes In RAM: 23 WITH END	
Registers To Copy: 60	

FURTHER ASSISTANCE ON E?

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

END

STACK AND ALPHA REGISTER ANALYSIS FOR DS

L-# INSTRUCTION	M	N	O	P	L	X	Y	Z	T
42♦LBL "DS"					1	x	y	z	t
43 SIGN					x	1			
44 RDN						y	z	t	1
45 RCL d						d	y	z	t
46 STO]			1 2 3 4 5 6 7						
47 SCI IND L						d'			
48 X<> d									
49 STO [	1 2 3 4 5' 6' 7								
50 "⊢*****"	6' 7' * * * * *	1 2 3 4 5'	6 7	1 2 3 4 5					
51 X<>]			1 2 3 4 5' 6' 7			67-----			
52 STO [	6 7								
53 "⊢**"	* *	1 2 3 4 5' 6 7	3 4 5' 6' 7	1 2 3 4 5 1 2					
54 X<> \		6 7				new d			
55 STO d						y	z	t	d
56 RDN									
57 CLA	Cleared	Cleared	Cleared	Cleared					
58 RTN					x	y	z	t	d

EP - ERASE PROGRAM MEMORY

Master Clear is the simplest and fastest way to clear RAM program memory. Unfortunately this also clears data registers, key assignments and SIZE information. EP provides a way to clear RAM programs that does not disturb any other part of RAM. Example 1:

```
 Before                              After
            CAT 1                              CAT 1

LBL"XYA"            XEQ EP
END      174 BYTES  --------→ .END.     245 BYTES
LBL"ABC"                                  PACK
END       49 BYTES            PACKING
.END.     22 BYTES                         CAT 1
                              .END.     07 BYTES
```

COMPLETE INSTRUCTIONS FOR EP

Just XEQ EP and all of the program memory will be replaced by packable nulls. PACKing is not really necessary unless you intend to increase the SIZE. Any programs read in or keyed in will merely overwrite the (invisible) nulls. EP can be regarded as a DELete function that ignores nonpermanent ENDs, stopping only when it reaches the permanent .END. . EP preserves the original contents of X, but alpha and the rest of the stack registers are lost.

EP has another feature that permits you to maintain in RAM a set of core "pseudo-ROM" routines which will not be erased by EP. For example, suppose you want to maintain the programs "AREA" and "VOL" in RAM at all times. Then you would set up program memory as show:

```
LBL"AREA"
  .
  .
END
LBL"VOL"
  .
  .
END
LBL"//"
RCL b
END
ENTER↑
ENTER↑
ENTER↑     6-byte
ENTER↑     buffer
ENTER↑
ENTER↑
```

LBL "//" is used to delimit the bottom of the program area to be preserved. The RCL b is decoded by EP to determine where to start erasing. Erasing will begin somewhere in the 6-byte buffer shown above.

MORE EXAMPLES OF EP

Example 2: Using the delimiter option, XEQ EP and PACK to erase all programs below LBL "//". PACKing must be done immediately after XEQ EP to restore CAT 1 linkage. Otherwise the next card read will overwrite all of program memory. Setting flag 14 overrides the delimiter feature and causes EP to clear all programs.

EP can be freely interrupted and SST'ed. It can also be downloaded and run if followed by LBL"//" RCL b END as explained above. This prevents EP from erasing itself.

```
    Before                                    After
              CAT 1                                    CAT 1
LBL"AREA"
END        32 BYTES                  LBL"AREA"
LBL"VOL"                             END        32 BYTES
END        21 BYTES      XEQ EP      LBL"VOL"
LBL"//"                  PACK        END        21 BYTES
END        11 BYTES   ----------→    LBL"//"
LBL"QRS"                             END        11 BYTES
END        40 BYTES                  .END.      13 BYTES*
.END.      29 BYTES
```

*NOTE: this area of program memory contains up to 6 leftover ENTER↑s.

If any assigned global labels are cleared by EP, the assignment bits in status registers ⊦ and e are not cleared. Pressing the formerly assigned key will crash the calculator for several seconds. To repair the key assignment bit map either use the ASN function to clear the formerly assigned keys or read in any program card in normal or USER mode.

Routine Listing For: EP

```
 79♦LBL "EP"          98 -
 80 SF 25             99 E3
 81 XEQ "//"         100 /
 82 FC?C 14          101 +
 83 FC?C 25          102 XROM "OM"
 84 GTO 14           103 .
 85 XROM "PD"        104 DSE Z
 86 3                105 XEQ 04
 87 -                106 "+-"
 88 7                107 X<> [
 89 /                108 STO IND Z
 90 INT              109 X<>Y
 91 GTO 13           110 X<> c
                     111 R↑
 92♦LBL 14           112 XROM "GE"
 93 XROM "C?"
                     113♦LBL 04
 94♦LBL 13           114 STO IND Z
 95 XROM "E?"        115 DSE Z
 96 16               116 GTO 04
 97 ST- Z            117 RTN
```

LINE BY LINE ANALYSIS OF EP

EP first tries to execute LBL"//". If successful, the result of RCL b is decoded (line 85) and changed to a register number (line 90). Otherwise, or if flag 14 was set, the curtain address (line 93) is used.

Whichever number is used forms the integer part of an iii.fff counter (line 101). The fff part is taken from the .END. location (line 95). Sixteen is subtracted from both iii and fff since OM will place the program/data curtain at 010_{16}. The LBL 04 section stores nulls down to the .END. location. Then a nonpacked permanent END code (line 106) is stored over the old .END. Restoring the original curtain, held in the stack, completes the program.

CONTRIBUTORS HISTORY FOR EP

EP was conceived and written by Keith Jarett (4360). See *PPC CALCULATOR JOURNAL*, V7N8P10b. Some bytes were saved by Roger Hill (4940). The use of the RAM delimiter (LBL"//") and flag 14 was suggested by William Cheeseman (4381).

TECHNICAL DETAILS

XROM: 10,31 | **EP** | SIZE: 000

Stack Usage:
- 0 T: 0
- 1 Z: 0
- 2 Y: temporary c
- 3 X: X
- 4 L: 0.E?-16

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: CLEARED
- 12 b: BYTES 2-6 CLEARED
- 13 c: USED BUT RESTORED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NOT USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage:
04:
05:
06: NOT USED
07:
08:
09:
10:
14: CLEARED
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 0

Global Labels Called:

Direct	Secondary
"//"	2D
PD	QR
E?	
C?	
GE	
OM	

Local Labels In This Routine:
04
13
14

Execution Time:
7.2 seconds + .13 seconds per register cleared.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? YES
Program File: VM
Bytes In RAM: 72
Registers To Copy: 60

Other Comments:

FINAL REMARKS FOR EP

EP will allow users to conveniently retain down-loaded and modified ROM routines in RAM. They need only be followed by the three line LBL"//" routine. EP can then be used instead of Master Clear to clear unwanted RAM programs.

CLRG, EP, and CK form a complete set of RAM clearing routines. They clear data registers, program memory, and key assignments, respectively.

FURTHER ASSISTANCE ON EP

Call William Cheeseman (4381) at (617) 235-8863.
Call Keith Jarett (4360) at (213 374-2583.

NOTES

EX - EXPONENT OF X

EX is an exponent function that operates much like a built-in function (such as SIGN).

Example 1: Isolate the exponent of 1/π. Key π 1/x, XEQ **EX**. The result is -1.

COMPLETE INSTRUCTIONS FOR EX

EX is used like any other single-argument mathematical function available in the HP-41. With an argument in the X register, XEQ **EX** will replace that argument with its exponent. The argument will be placed in LSTX and the rest of the stack will be left unchanged.

There are times when one wishes to extract, for observation or manipulation, the exponent of a number stored in the X register of the HP-41. For example, one might wish to extract the number 95 from the number 4.526114387 95 (4.526114387 times 10 to the 95th) in a routine to perform operations with numbers too large or too small to be handled normally by the HP-41.

At first glance, this seems like a trivial task, since executing a LOG function followed by an INT function seems to do the job. However, LOG, INT yields incorrect results with numbers having negative exponents. While routines may be created which properly deal with this deficiency, another more difficult problem must still be faced. This more difficult problem is the result of rounding or truncation error inherent in any numeric device having limited resolution. Trying LOG, INT on 9.9999999 23 will produce a result of 24. While 10 to the 24th is approximately equal to 9.9999999 23, 24 is not an acceptable result to an **EX** routine. (A calling program which is unaware of the rounding which has taken place might reconstruct the original number to be 9.9999999 24!) This problem, too, can be circumvented with standard non-synthetic programming techniques, but the resulting routine must necessarily by long and slow (LOG alone takes approximately 250ms).

The PPC ROM **EX** routine uses byte manipulation in the ALPHA register to extract the exponent of a number initially in the X register. This technique avoids rounding errors as well as time consuming mathematical functions. A new problem arises using this approach, however. Internally, the HP-41 stores the exponent of a number in a three digit unsigned configuration. Exponents such as 00, 23, or 99, for example, are stored respectively as 000, 023, and 099. However, negative exponents, such as -23 and -99 are stored as (1000-23) or 977, and (1000-99) or 901, respectively. Simply extracting the byte which represents the two digit part of the exponent is not adequate. Notice must be taken of the sign of the (two digit) exponent, and, if negative, 100 must be subtracted from it.

Routine Listing For: **EX**

```
14•LBL "EX"
15 CLA
16 X≠0?
17 "θΩ"
18 INT
19 X≠0?
20 CLA
21 RDN
22 LASTX
23 X<> [
24 ASHF
25 "⊢•▵"
26 ST- [
27 X<> [
28 RTN
```

LINE BY LINE ANALYSIS OF EX

The routine begins by clearing the Alpha register, thus setting registers M, N, O, & P to zero. The contents of M will eventually contain either zero or 100 depending upon whether the magnitude of the input is (zero or at least 1) or (non-zero but less than 1), respectively. The routine proceeds by testing the X register and loading M with "θΩ" if X is non-zero. "θΩ" are hexadecimal bytes 10 and 11, respectively. When loaded into the M register in this fashion, these bytes yield the non-normalized number 0.000000001 11 which if normalized equals 1.000000000 02 (or 100). If the integer part of the argument is non-zero then the argument must be equal to or greater than 1 so the M register is once again set to zero with CLA. (Note that the HP-41 has an INT function which takes the integral part of the absolute value of the argument, then attaches the sign of the argument to the result.) Having now placed the correct value, 0 or 100, in the M register, the routine swaps X for M where ASHF strips the argument of everything but its two digit exponent, ee. Appending "•▵", which are hexadecimal bytes 00 and 08, respectively, to the exponent yields the non-normalized number 0.000000ee0 08 which if normalized equals e.e00000000 01. With the equivalent of 0 or 100 in X, STO-M performs two functions: subtracting a non-normalized 100 from the exponent, if required, and normalizing the answer in either case. The answer, in M, is finally swapped with X and the routine ends.

CONTRIBUTORS HISTORY FOR EX

Dave Kaplan (3678) wrote **EX**, superseding a synthetic version by Roger Hill (4940) and all versions which used the LOG function. **EX** is much faster than all previous exponent routines.

FURTHER ASSISTANCE ON EX

Call David R. Kaplan (3678) at (703) 250-6621.
Call Keith Kendall (5425) at (801) 967-8080.

TECHNICAL DETAILS

XROM: 10,27	**EX**	SIZE: 000

Stack Usage:	Flag Usage: NONE USED
0 T: UNCHANGED	04:
1 Z: UNCHANGED	05:
2 Y: UNCHANGED	06:
3 X: exponent of X	07:
4 L: X	08:
Alpha Register Usage:	09:
5 M: CHANGED	10:
6 N: CLEARED	
7 O: CLEARED	
8 P: CLEARED	25:
Other Status Registers:	Display Mode: UNCHANGED
9 Q:	
10 ⊢:	
11 a:	Angular Mode: UNCHANGED
12 b: NONE USED	
13 c:	
14 d:	Unused Subroutine Levels:
15 e:	6
ΣREG: UNCHANGED	Global Labels Called:
Data Registers: NONE USED	Direct / Secondary
R00:	NONE / NONE
R06:	
R07:	
R08:	
R09:	
R10:	
R11:	Local Labels In This Routine:
R12:	NONE

Execution Time: .5 seconds.

Peripherals Required: NONE

Interruptible? YES	Other Comments:
Execute Anytime? NO	
Program File: VM	
Bytes In RAM: 28	
Registers To Copy: 60	

NOTES

F? - FREE REGISTER FINDER

F? gives a count of the number of registers available for programs and/or key assignments. The result of XEQ F? will normally equal the number of registers displayed on line 00 of a RAM program (00 REG nn or .END. REG nn), except that F? will show an extra 1/2 register available if the top key assignment register has its right half vacant. F? provides a programmable equivalent to the manual procedure of switching to PRGM mode at line 00.

Example 1: MASTER CLEAR and XEQ F? . The result is 46 registers available, the same as the program mode display shows. Now ASN F? to a key and execute it. You'll see 45 registers available. Actually 45.5 are available, but F? , like A? and LF , are not fully compatible with ASN. See A? and the MK background for details. If you XEQ PK then re-execute F? the full 45.5 registers will be indicated.

COMPLETE INSTRUCTIONS FOR F?

Just XEQ F? to get a count of the number of free registers. This count may be 1/2 register too low under the same conditions that A? may give a count that is 1/2 register too high (see A? for details). F? will always give the correct count after PK . The output of F? is related to those of E? and A? by the formula F? = E? - A? - 192 .

Although F? can be interrupted or single-stepped, you should let it run to completion to avoid leaving the curtain at absolute address 16. F? uses the alpha register and the whole stack. A temporary c register from OM is left in Y, Z, and T. Flag 10 will be left set if the result is not an integer.

LINE BY LINE ANALYSIS OF F?

Line 196 produces a pointer of the form bbb.eee to the free register block, relative to a curtain address of 16, with flag 10 set if the first register of that block has a key assignment in its left half. See LF for details. The bbb part of this pointer is one plus the location of the topmost full assignment register, while the eee part is the location of the .END. minus one. Thus the number of free registers is eee - bbb + 1 if flag 10 is clear, or eee - bbb + 1/2 if flag 10 is set. This calculation is performed on lines 197-209. The LBL 11 entry point is provided for MK's "REG FREE:" display.

CONTRIBUTORS HISTORY FOR F?

F? was written by Roger Hill (4940) as an additonal entry point to his group of key assignment programs.

FURTHER ASSISTANCE ON F?

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

Routine Listing For: F?

```
195♦LBL "F?"        202 *
196 XROM "LF"       203 X<>Y
                    204 .5
197♦LBL 11          205 FC? 10
198 INT             206 SIGN
199 LASTX           207 -
200 FRC             208 -
201 E3              209 END
```

TECHNICAL DETAILS

XROM: 10,04 | **F?** | SIZE: 000

Stack Usage:
- 0 T: temporary c
- 1 Z: temporary c
- 2 Y: temporary c
- 3 X: result
- 4 L: used

Flag Usage: ONLY FLAG 10 IS ALTERED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10: SET IF F? NOT AN INTEGER
- 25:

Alpha Register Usage:
- 5 M:
- 6 N: ALL LOST
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED BUT RESTORED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 3

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R06:
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Global Labels Called:

Direct	Secondary
LF	E?
	OM
	2D

Local Labels In This Routine: 11

Execution Time: 9.1 second. (For 16 assignment registers)

Peripherals Required: NONE

Interruptible? YES

Execute Anytime? YES

Program File: MK

Bytes In RAM: 26 WITH END

Registers To Copy: 61

Other Comments:

CONTINUED FROM PAGE 135

MEM	QUAN	ORD
2795	1	969
2796	1	140
2801	2	271
		1394
2807	1	716
2814	1	2352
2816	2	566
		567
2826	1	443
2831	1	1115
2838	1	292
2843	1	41
2846	1	106
2858	1	2244
2860	1	33
2864	1	500
2865	1	417
2870	1	636
2884	1	1118
2887	1	390
2895	1	1528
2896	1	737
2899	1	849
2900	1	336
2903	1	1631
2907	1	261
2910	1	2144
2912	1	1435
2915	1	1781
2916	1	2
2924	1	181
2927	1	666
2945	1	984
2946	1	1352
2952	1	327
2957	1	1180
2960	1	1561
2961	1	1101
2972	1	348
2975	1	651
2978	1	1202
2980	1	513
2997	2	436
		1659
2998	2	224
		225
3005	1	1612
3009	1	2202
3022	1	335
3047	1	1906
3051	1	1942
3059	1	1279
3060	1	1997
3065	1	201
3068	1	267
3073	1	2002
3074	1	24
3078	1	1946
3080	1	552
3086	1	2100
3087	1	1788
3092	1	899
3101	1	351
3104	1	10
3106	1	682
3107	1	163
3110	1	613
3117	1	1396
3134	1	1483
3135	1	794

MEM	QUAN	ORD
3151	1	377
3152	1	532
3176	1	32
3179	1	437
3192	1	434
3193	1	671
3197	2	172
		2492
3205	1	1990
3209	2	607
		2267
3211	1	754
3217	1	441
3222	1	278
3234	2	948
		949
3237	1	976
3240	1	526
3245	1	1035
3246	1	612
3254	1	45
3259	1	2081
3274	1	280
3288	1	649
3294	1	1579
3301	1	114
3304	1	1158
3305	2	153
		154
3310	1	266
3321	1	1339
3324	1	1559
3332	2	620
		621
3335	1	1428
3337	1	627
3345	1	2051
3347	1	549
3350	1	2355
3353	1	178
3362	1	1498
3374	1	2137
3376	1	972
3380	1	1467
3382	1	1719
3385	1	306
3386	1	1701
3388	1	2297
3390	1	1740
3397	1	1544
3404	1	1658
3411	1	2269
3413	1	270
3417	1	878
3421	1	216
3426	1	1046
3433	1	1249
3438	1	741
3452	1	1520
3454	1	1492
3456	2	535
		536
3462	1	815
3465	1	299
3466	1	1326
3480	1	320
3482	1	252
3485	1	28
3489	2	401
		402
3491	1	1131

MEM	QUAN	ORD
3494	1	2155
3498	1	2046
3500	1	9
3501	1	1919
3502	1	77
3508	1	653
3514	1	182
3521	1	2224
3522	1	654
3524	1	2446
3527	2	157
		2050
3528	1	945
3529	1	445
3531	1	1356
3536	1	2052
3543	1	783
3545	1	2480
3562	1	363
3563	1	506
3569	1	1898
3571	1	444
3572	1	1183
3573	1	405
3575	1	466
3582	1	538
3591	1	61
3592	1	2250
3595	1	780
3601	2	829
		830
3602	1	1116
3604	1	501
3605	1	1562
3606	1	19
3610	1	2282
3615	1	1398
3627	1	1004
3637	1	935
3638	1	1806
3648	1	1611
3649	1	749
3660	2	254
		1912
3663	1	1253
3665	1	1176
3666	1	11
3667	1	1484
3668	1	2057
3676	2	1238
		1239
3678	1	1283
3680	1	1586
3695	1	2131
3706	2	1064
		1065
3716	1	1478
3727	1	380
3729	3	591
		592
		593
3733	1	1232
3735	2	1449
		1450
3736	1	308
3749	1	2170
3752	1	1765
3754	1	2356
3759	1	381
3771	1	485
3795	1	2066

MEM	QUAN	ORD
3796	1	1310
3801	1	2366
3803	1	323
3815	1	641
3825	1	249
3832	4	926
		927
		928
		929
3833	1	1141
3838	1	528
3839	1	2123
3842	1	1807
3866	1	816
3875	1	1459
3878	2	1036
		1037
3882	1	467
3897	1	1662
3915	1	143
3917	1	1949
3920	1	1420
3922	1	646
3924	1	1587
3927	1	1463
3932	1	2348
3933	1	1385
3934	2	1972
		1973
3935	1	330
3936	1	1079
3941	1	1211
3942	1	2217
3948	1	1246
3949	1	1063
3951	1	674
3956	1	1699
3957	2	931
		932
3965	1	75
3966	1	1000
3970	1	1812
3972	1	1864
3981	2	510
		511
3982	1	1173
3983	1	997
3988	1	1570
3989	1	1728
3992	1	2427
3994	1	385
3996	1	212
3997	1	391
3999	1	333
4002	1	1971
4004	1	544
4010	1	797
4015	1	1393
4016	1	576
4021	1	527
4023	1	155
4026	1	2098
4031	1	233
4037	1	1291
4042	1	1309
4044	1	1123
4047	2	1295
		2257
4049	1	1121
4057	1	189
4058	1	1542

MEM	QUAN	ORD
4061	1	104
4063	1	209
4065	1	1422
4068	1	392
4075	1	125
4076	1	903
4077	1	117
4078	1	1106
4089	1	933
4092	1	207
4093	2	1635
		1636
4098	1	95
4100	1	2261
4102	1	608
4108	1	1098
4115	1	555
4116	1	256
4120	1	1297
4123	1	2007
4124	1	1289
4128	1	1086
4129	1	422
4130	1	1866
4131	1	442
4132	1	253
4133	1	2337
4137	1	171
4144	1	523
4146	1	1625
4154	1	1571
4158	5	1159
		1160
		1161
		1162
		1163
4163	1	1391
4164	1	1043
4168	1	1715
4170	3	1092
		1093
		1094
4171	1	2097
4176	1	1287
4181	2	387
		388
4185	1	46
4188	1	293
4191	1	451
4196	1	1563
4197	1	492
4198	1	694
4203	1	982
4204	1	2106
4206	1	1597
4213	1	1555
4218	1	1890
4223	1	2008
4227	1	1538
4229	1	1363
4231	1	259
4234	1	304
4237	1	85
4239	1	1142
4240	1	1156
4248	2	118
		119
4255	2	738
		739
4258	2	764
		765
4266	1	1165

MEM	QUAN	ORD
4268	2	1491
		1499
4269	1	97
4272	1	947
4277	1	373
4281	2	1051
		1052
4291	1	2005
4292	1	524
4293	3	135
		136
		137
4294	1	329
4295	1	522
4296	3	169
		1343
		2487
4299	2	53
		1205
4300	1	979
4303	2	622
		623
4304	2	173
		174
4305	1	58
4311	2	1493
		1494
4314	1	667
4315	1	257
4316	1	183
4317	1	462
4319	1	2460
4324	2	90
		91
4325	1	2407
4331	1	657
4333	1	355
4337	1	611
4338	1	109
4340	1	2061
4341	1	846
4342	1	1830
4343	1	322
4344	1	507
4346	2	468
		2193
4347	2	858
		859
4348	1	482
4352	1	1357
4353	1	1419
4359	1	452
4360	4	556
		1861
		1862
		1863
4361	1	2438
4362	1	1112
4363	1	778
4365	1	725
4367	1	369
4371	1	2335
4375	1	277
4378	1	476
4379	1	865
4380	1	543
4381	1	30
4382	1	525
4390	1	1127
4391	1	240
4394	7	1778
		1779
		1884

MEM	QUAN	ORD
		1885
		1886
		1887
		1888
4398	1	229
4399	1	2126
4402	1	1726
4403	1	112
4404	1	455
4405	1	1074
4406	1	15
4416	1	777
4417	1	1832
4419	1	1573
4420	1	1048
4421	1	1049
4423	1	991
4425	1	1433
4426	1	305
4427	1	1252
4430	1	345
4431	1	2092
4432	1	236
4433	1	1589
4435	1	676
4439	1	362
4440	1	1517
4442	1	1031
4443	1	1605
4444	2	1088
		1594
4448	1	1603
4449	1	2190
4456	3	2197
		2198
		2199
4457	2	1039
		1040
4461	1	1333
4464	1	946
4465	1	1060
4467	1	1578
4470	1	1682
4471	1	1543
4472	1	1407
4474	1	554
4475	1	950
4477	1	966
4478	1	407
4479	1	113
4480	1	1190
4481	3	1746
		1747
		1748
4486	2	672
		673
4492	1	2439
4493	1	503
4494	1	842
4495	1	1125
4499	1	533
4500	1	1508
4501	1	668
4502	1	744
4503	1	1129
4504	1	358
4506	1	1254
4509	1	2147
4511	1	307
4513	2	1171
		1172
4515	1	2218

APPENDIX F CONTINUED ON PAGE 165

FD - FIRST DERIVATIVE

This routine will approximate the first derivative of a function at a point in one of two ways. A quick four point polynomial estimate may be made using a step size that is provided by the user, or a more precise adaptive procedure may be used which automatically searches for the optimal step size. In the adaptive routine setting a flag allows the user to view convergence of the optimal step size. The adaptive procedure delivers an error estimate along with the derivative, but 6 1/2 decimal digits is all that can be trusted in any case. Both versions of FD sample on only one side of the evaluation point so that FD may be used for one-sided derivatives. Discontinuities or singularities may be avoided by selecting an increment with the appropriate sign. The routine may also be used to compute partial derivatives.

Example 1: Use FD to approximate f'(2) where

$$f(X) = 3X^3 - 4X^2 + 5X + 6$$

First establish SIZE 020 which should be a sufficient size for almost all derivatives. The function must be programmed as a subroutine. The function f(X) will take its argument X from a register pointed to indirectly by register R11. For this example we will use register R18 to hold X and will store the number 18 in R11. The output from the function subroutine, namely f(X), is to be left in the X-register. For this example the following f(X) routine may be programmed in RAM program memory.

```
01*LBL "FX1"
02 RCL IND 11
03 ENTER↑
04 ENTER↑
05 ENTER↑
06 3
07 *
08 4
09 -
10 *
11 5
12 +
13 *
14 6
15 +
16 RTN
```

The name of the global label "FX1" should be stored in R10. Go into alpha mode and key "FX1" ASTO 10. Set flag F09 to select the quick approximation method for this example. After storing 18 in R11 store a step size of 0.1 in register R12. Store the value of X (=2) in R18. Then to calculate the derivative XEQ "FD". In this example the exact answer of 25 is given because the polynomial formula is exact for polynomials of degree three or less.

COMPLETE INSTRUCTIONS FOR FD

1) First select a SIZE which must be a minimum of 018 depending on how many additional registers the function requires.

2) The function must be programmed as a subroutine in RAM program memory with a global label name of six or less characters. The input to this subroutine, namely X, will come from a data register pointed to by R11. The output from this subroutine, namely f(X), should be left in the X-register. f(X) must not modify registers R10 through R17, flag F09 or flag F10.

3) Store the global label function name from step 2) in R10.

4) Store the number of the register to hold X in the f(X) subroutine in R11. This register may normally be any register other than any of R10 through R17.

5) If the calculation of f'(a) is desired, store the initial X-value a in the register named in step 4).

6) Store an initial step size in register R12.

7) Select the quick approximation or the adaptive procedure by the status of flag F09. If F09 is set the quick approximation will be performed. If F09 is clear the adaptive procedure will be selected. (NOTE: If you use the adaptive option the initial step size stored in R12 must be large enough for the algorithm to iterate at least three times before terminating).

8) If the adaptive procedure was selected in step 7) then an additional option is to view the convergence of the optimal step size. Set flag F10 to display the values. If F10 is clear only the final answer will be displayed.

9) XEQ "FD". An estimate of the first derivative will be left in the X-register. In addition, if the adaptive procedure was used an error estimate will appear in the Y-register.

MORE EXAMPLES OF FD

Example 2: Use FD to find the gradient of F(x,y)= $[x + \ln(y)]^2$ at the point P(2,1).

The gradient is simply the vector whose components are the partial derivatives of F(x,y) so this example will require the calculation of two partial derivatives. Create F(x,y) in program memory with x in register R20 and y in register R21.

```
01*LBL "FXY2"
02 RCL 20
03 RCL 21
04 LN
05 +
06 X↑2
07 RTN
```

The following auxiliary program will be used to set up and execute FD.

```
08*LBL "GRA"
09 "FXY2"
10 ASTO 10
11 20
12 STO 11
13 .1
14 STO 12
15 2
16 STO 20
17 1
18 STO 21
19 CF 09
20 SF 10
21 XROM FD
```

```
22 STO 22
23 ISG 11
24 ABS
25 XROM FD
26 STO 23
27 RTN
```

After executing "GRA" the gradient will be in memory with dF/dX in R22 and dF/dY in R23.

R22: 4.000000000   R23: 3.999999379

Example 3: Use FD to find the left and right derivatives at t=0 for the function:

$$F(t) = \begin{cases} e^t - t & \text{if } t<=0 \\ (t+1)^3 & \text{if } t>0 \end{cases}$$

Note that F(t) is continuous at t=0. The following routine will serve as the F(t) subroutine.

```
01*LBL "FT3"
02 RCL 20
03 X>0?
04 GTO 01
05 E↑X
06 LAST X
07 -
08 RTN
09 LBL 01
10 1
11 +
12 3
13 Y↑X
14 RTN
```

Store the following values.

```
R10: "FT3" = the name of the function
R11: 20    = pointer to t
R12: .01   = increment value (step size)
R20: 0     = t value
```

Set flag F09 for the quick polynomial evaluation and XEQ "FD". Since the increment value was positive the right hand derivative is the value returned. See 3.000000000. To calculate the left-hand derivative simply change the increment value in R12 to -.01 and XEQ "FD" a second time. See -0.000000350 returned.

If greater precision in the left-hand derivative is desired, perform the same procedure with an initial increment of -.1 and clear flag F09 so the adaptive routine will be used. In this case the left hand derivative is -0.000000060. The true answer is 0.

Example 4: $F(U) = U^4 + 4U^3 + 12U^2 + 24U + 24$
Use FD to find F'(1).

Create F(U) in program memory and create an auxiliary program to set up and execute FD.

```
01*LBL "FU4"          18*LBL "DFU"
02 RCL 20             19 STO 20
03 ENTER↑             20 SF 09
04 ENTER↑             21 XROM FD
05 ENTER↑             22 STO 21
06 4                  23 -1
07 +                  24 STO*12
08 *                  25 XROM FD
09 12                 26 RCL 21
10 +                  27 +
11 *                  28 2
12 24                 29 /
13 +                  30 RTN
14 *
15 24
16 +
17 RTN
```

This program takes the average derivative for positive and negative step sizes and is exact for a polynomial of degree 4 or less, regardless of the step size. Key "FU4" ASTO 10   20 STO 11   1 STO 12   To calculate F'(1) key in 1 and XEQ "DFU". The true answer is 64. To calculate F'(U) at any value of U, key in U and XEQ "DFU". As another example, F'(2)=152.

Example 5: Find the extreme value that is closest to zero of the quartic equation given in Example 4.

FD is compatible with the root finder SV since the two programs use no registers in common. To find where the extreme is, we will solve for a zero in the first derivative. To do this create the following program. We assume FU4 is still in program memory from the previous example. Change lines 18, 19 & 20 to the following:

```
18*LBL "DD"
19 STO IND 11
20 GTO FD
```

Next, store the following values:

```
R06 "DD"  = name of function to be solved
R10 "FU4" = name of function to be differentiated
R11  20   = pointer to U
R12 .01   = increment value
```

Set flag F09 to select quick polynomial estimate. FIX 7   .1 ENTER↑ 0   XEQ "SV". The location of the extremum will be returned in about 1 minute, see -1.5960722, where the actual derivative is about 0.00001.

FURTHER DISCUSSION OF FD

Besides the obvious application of finding the first derivative of a function, FD is useful for partial differentiation, where the gradient is needed for higher-dimensional root finding schemes, and in optimization problems where compatibility with SV is a useful feature. In a gradient problem it is a good idea to place the variables in consecutive registers so that they may be recalled via an ISG control word.

Generally, when a function is known, it is best to use calculus to create a derivative program; sometimes, however, a program must process a user defined function and FD provides the required derivative. FD is also useful when the function to be differentiated is complicated, or when you need a numerical check on the answer you get by using calculus.

The differentiator is not completely automatic because the user is required to provide an initial increment value for each variable. The reference paper provides an algorithm for automatically determining an initial increment, and the algorithm has been coded for the HP-41C by Harry Bertucelli (3994). However, the algorithm was too long to include in the ROM.

FORMULAS USED IN FD

The quick 4-point polynomial approximation uses the following formula.

$$f'(X) = [-11f(X) + 18f(X+h) - 9f(X+2h) + 2f(X+3h)]/6h$$

where h is the current step size.

The above formula may be found in the HANDBOOK OF MATHEMATICAL FUNCTIONS, table 25.2. The formula samples the user's function on only one side of the point where the derivative is being evaluated; on the right if h>0 and on the left if h<0. Thus a discontinuity or singularity may be avoided by selecting an increment with the appropriate sign. The formula will be exact for polynomials of degree 3 or less, and the average of estimates will be exact for polynomials of degree 4 or less. For most functions, however, the error of the polynomial is of the order of h to the 4th power.

The adaptive routine works by evaluating the polynomial formula for successively smaller h values. The sequence of estimates D_{i+1} should be monotonic and the sequence $|D_{i+1} - D_i|$ should be monotonic and decreasing. If either monotonicity condition is violated, numerical truncation is causing error in D_{i+1} and D_i is delivered to the user as the final output. $|D_{i+1} - D_i|$ is placed in the stack as an estimate of the error. Note however that 6 1/2 digits is the most you can depend on regardless of the error estimate. The final h value is left in R12 so that any subsequent derivative evaluations at nearby points may be made by the quick polynomial formula using the nearly optimum increment value h.

Routine Listing For: FD

```
124♦LBL D            161 X<> L
125♦LBL "FD"         162♦LBL 07
126 FS? 09           163 R↑
127 GTO 08           164 .7
128 17               165 ST/ 12
129 XROM "SD"        166 CLX
130 SCI 1            167 17
131 2 E-3            168 XROM "RD"
132 STO 14           169 RTN
133♦LBL 05           170♦LBL 08
134 RCL 12           171 .
135 .7               172 STO 13
136 *                173 XEQ IND 10
137 RND              174 11
138 STO 12           175 XEQ 09
139 XEQ 08           176 -18
140 ENTER↑           177 XEQ 09
141 X<> 16           178 9
142 -                179 XEQ 09
143 ENTER↑           180 ST+ X
144 FS? 10           181 RCL 13
145 VIEW X           182 -
146 X<> 15           183 RCL 12
```

```
147 ISG 14           184 3
148 GTO 05           185 *
149 LASTX            186 ST- IND 11
150 RDN              187 ST+ X
151 X=0?             188 /
152 GTO 07           189 ENTER↑
153 /                190 RTN
154  E               191♦LBL 09
155 X<>Y             192 *
156 X<0?             193 ST+ 13
157 GTO 06           194 RCL 12
158 X<Y?             195 ST+ IND 11
159 GTO 05           196 GTO IND 10
160♦LBL 06           197 END
```

LINE BY LINE ANALYSIS OF FD

Lines 124-127 start the program and a test of flag F09 is made to determine whether the quick polynomial method is to be made.

Lines 128-132 help initialize the program. The display mode is saved in R17 and FD then changes to a SCI 1 display mode. The 0.002 stored in R14 is a loop counter!

Lines 133-148 serve both as part of the initialization and as part of the main loop in the program. Since D_i and $|D_i - D_{i-1}|$ are unknown initially, comparisons on these values are avoided on the first three passes by means of the ISG test at lines 147 and 148. Rounding to SCI 1 at steps 130 and 137 is used to prevent significant digits of h from underflowing. Line 139 is the subroutine call to the 4-point polynomial estimate which leaves two copies of D_i in X and Y.

Lines 149-159 continue the main loop. After recalling LAST X at line 149 the stack content may be assumed to be:

X: D_{i-1} Y: $D_{i-1}-D_{i-2}$ Z: D_i-D_{i-1} T: D_i

The monotonicity of both D_i and $|D_i-D_{i-1}|$ is tested at steps 153 through 159 by using the ratio:

$$r = [D_{i+1}-D_i]/[D_i-D_{i-1}]$$

If r<0 or if r>=1 then the routine ends. If 0<=r<1 then the main loop is iterated again.

Lines 160-163 ensure that the stack contents at line 163 are:

X: D_{i-1} Y: $D_{i-1}-D_{i-2}$

Lines 164-166 restore the previous h value without lifting the stack.

Lines 167-169 restore the original display mode at the time FD was called and end the routine.

Lines 170-196 perform the 4-point polynomial estimate by computing the 4-point formula. Line 186 ensures that the original x is restored in R11 since part of these lines add multiples of h to x. Two copies of D_i are in X and Y when the routine ends at line 190.

REFERENCES FOR FD

1. Stepleman and Winarsky, "ADAPTIVE NUMERICAL DIFFERENTIATION", Mathematics of Computation, Vol. 33 No. 148 pp. 1257-1264 (October 1979).

2. Abramowitz and Stegun, "HANDBOOK OF MATHEMATICAL FUNCTIONS", National Bureau of Standards, Applied Mathematics Series, US Department of Commerce 55

3. John Kennedy "PPC Calculator Journal," ROM Progress V7N9P14.

CONTRIBUTORS HISTORY FOR FD

The FD routine was first suggested by Harry Bertucelli (3994) who had written a program that was too lengthy to include in the ROM. A less ambitious attempt resulted in the FD program. In addition to Harry credit goes to Richard Schwartz (2289), Ron Knapp (618) and Martin Sitte (6224) who made or suggested improvements in the FD routine. Richard Schwartz worked on the documentation for FD.

FINAL REMARKS FOR FD

A future version of FD may be a full implementation of the routine first developed by Harry Bertucelli (3994).

FURTHER ASSISTANCE ON FD

Harry Bertuccelli (3994) Work: (213) 648-7000
Home: (213) 846-6390 (after 8PM)

Richard Schwartz (2289) phone: (213) 447-6574 eve.

NOTES

TECHNICAL DETAILS

XROM: 20, 11 — **FD** — SIZE: 018 minimum

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: set=quick approx. clear=adaptive proc.
- 10: set=display approx. clear=no display
- 25: not used

Display Mode:
SCI n recommended

Angular Mode:
not used, but may depend on function

Unused Subroutine Levels:
4

ΣREG: not used

Data Registers:
- R00: not used
- R06: not used
- R07: not used
- R08: not used
- R09: not used
- R10: function LBL name
- R11: pointer to x
- R12: increment in x
- R13: scratch, not to be used by function
- R14: loop counter
- R15: difference in previous two approximations
- R16: previous approximation
- R17: saves display mode

Global Labels Called:

Direct	Secondary
SD	
RD	

Local Labels In This Routine:
D 05, 06, 07, 08, 09

Execution Time: variable, depends on f(x)

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: IG

Bytes In RAM: 123

Registers To Copy: 43

Other Comments:

FI - FINANCIAL CALCULATIONS

This is a complete financial program that uses the top two rows of keys to either input or solve for the five standard financial values; n, %I, PV, PMT, FV.

This highly accurate program extends the capabilities of previous HP financial calculators and programs by adding two new parameters.

1. "CF" The Compounding Frequency can be specified (including continuous compounding) and may be different than the payment frequency.

2. "PF" The Payment Frequency can be specified and may be different from the compounding frequency.

This added facility simplifies the solution of some complex financial problems that are difficult to solve via the standard financial calculator or program. Canadian and European style mortgage problems can now be handled in a simple straightforward manner.

A Beginning/End of period switch is provided and a status display function allows the user to determine the current state of toggle controlled functions. The "CLEAR" financial register function incorporates default parameters that permit the user to operate the program in the same manner as typical financial calculators or programs that do not include the facility to specify different compounding and payment periods. Standard financial sign conventions are used (money paid out is negative, money received is positive).

The LN1+X and E↑X-1 functions are used in compounding routines instead of Y↑X, resulting in more precise answers than are produced by most financial programs and calculators.

FI

BACKGROUND FOR FI

Time Value of Money:

If you borrow money you can expect to pay rent or interest for its use; conversely you expect to receive interest on money you loan or invest. When you rent property, equipment, etc., rental payments are normal; this is also true when renting or borrowing money. Therefore, money is considered to have a "time value". Money available now, has a greater value than money available at some future date, because of its rental value or the interest that it can produce during the intervening period.

Simple Interest:

If you loaned $800 to a friend with an agreement that at the end of one year he would repay you $896, the "time value" you placed on your $800 (principal) was $96 (interest) for the one year period (term) of the loan. This relationship of principal, interest and time (term) is most frequently expressed as an Annual Percentage Rate (APR). In this case the APR was 12.0% [(96/800)100]. This example illustrates the four basic factors involved in a simple interest case. The time period (one year), rate (12.0% APR), present value of the principal ($800) and the future value of the principal including interest ($896).

Compound Interest:

In many cases the interest charge is computed periodically during the term of the agreement. For example, money left in a savings account earns interest that is periodically added to the principal and in turn earns additional interest during succeeding interest periods. The accumulation of interest during the investment period represents compound interest. If the loan agreement you made with your friend had specified a "compound interest rate" of 12.0% (compounded monthly) the $800 principal would have earned $101.46 interest for the one year period. The value of the original $800.00 would be increased by 1% the first month to $808.00 which in turn would be increased by 1% to $816.08 the second month, reaching a future value of $901.46 after the twelfth iteration. The monthly compounding of the nominal annual rate (NAR) of 12% produces an effective Annual Percentage Rate (APR) of 12.683% [(101.46/800)100]. Interest may be compounded at any regular interval; annually, semiannually, monthly, weekly, daily, even continuously (a specification in some financial models).

Periodic Payments:

When money is loaned for longer periods of time it is customary for the agreement to require the borrower to make periodic payments to the lender during the term of the loan. The payments may be only large enough to repay the interest, with the principal due at the end of the loan period (an interest only loan), or large enough to fully repay both the interest and principal during the term of the loan (a fully amortized loan). Many loans fall somewhere between, with payments that do not fully cover repayment of both the principal and interest. These loans require a larger final payment (balloon) to complete their amortization. Payments may occur at the beginning or end of a payment period. If you and your friend had agreed on monthly repayment of the $800 loan at 12.0% NAR compounded monthly, twelve payments of $71.08 for a total of $852.96 would be required to amortize the loan. The $101.46 interest from the annual plan is more than the $52.96 under the monthly plan because under the monthly plan your friend would not have had the use of $800 for a full year.

Financial Transactions:

The above paragraphs introduce the basic factors that govern most financial transactions; the time period, interest rate, present value, payments, and the future value. In addition, certain conventions must be adhered to; the interest rate must be relative to the coumpounding frequency and payment periods, and the term must be expressed as the total number of payments (or compounding periods if there are no payments). Loans, leases, mortgages, annuities, savings plans, appreciation, and compound growth are among the many financial problems that can be defined in these terms. Some transactions do not involve payments, but all of the other factors play a part in "time value of money" transactions. When any one of the five (four- if no payments are involved) factors is unknown, it can be derived from formulas using the known factors. This is the function of the FI financial program.

Problem Solving Preliminaries:

Diagram or visualize the positive and negative cash flows and their timing. (See cash flow diagrams)

Clear the financial registers by pressing e , unless the problem is a continuation or minor change from the preceding problem. Note that clearing also sets the compounding and payment frequency values to 1.

Check the mnemonic status code for applicability to the current problem and change it by pressing c and/or d if necessary. The mnemonic status codes are:

CB = Continuous compounding and Beginning of period payments.
CE = Continuous compounding and End of period payments.
DB = Discrete compounding and Beginning of period payments.
DE = Discrete compounding and End of period payments.

Specify the compounding and payment frequencies by entering appropriate values and pressing H and/or I.

Generalized Cash Flow Diagrams:

Selection of the proper parameters and signs of the factors in a specific financial transaction can often be aided by constructing a cash flow diagram similar to the examples below:

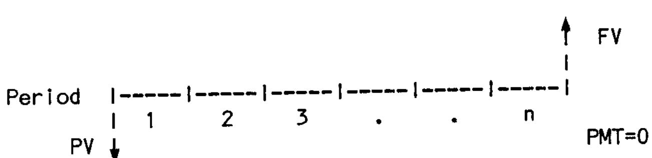
Standard Financial Conventions are:

Money RECEIVED is a POSITIVE value and is represented by an arrow above the line.

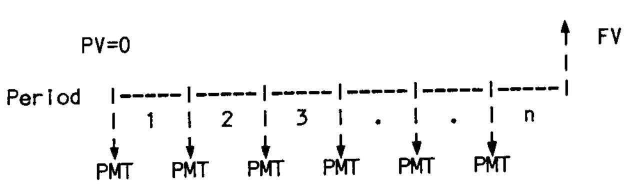
Money PAID OUT is a NEGATIVE value and is represented by an arrow below the line.

If payments are a part of the transaction the number of payments must equal the number of periods (n).

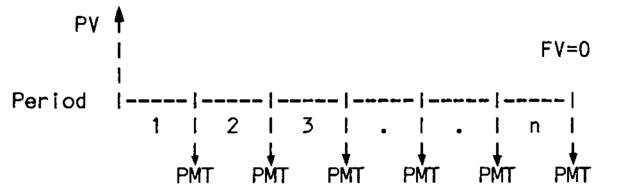
Payments may be represented as occuring at the end or beginning of the periods.



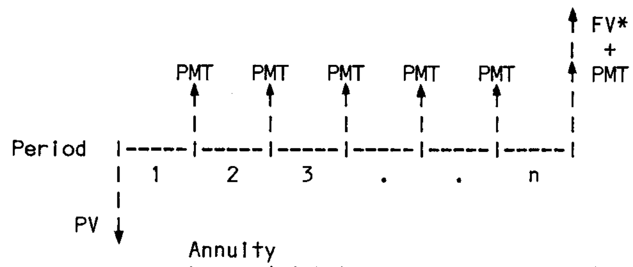
Appreciation
Depreciation
Compound Growth
Savings Account



Annuity (series of payments)
Pension Fund
Savings Plan
Sinking Fund



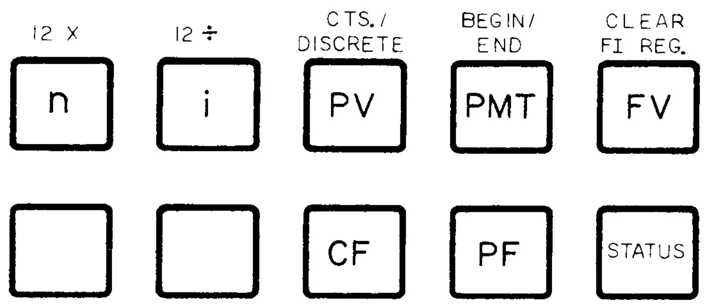
Amortization
Direct Reduction Loan
Mortgage (fully amortized)



Annuity
Lease (with buy back or residual)*
Loan or Mortgage (with balloon)*

COMPLETE INSTRUCTIONS FOR FI

By manually keying GTO "FI" the program pointer is in the ROM and the following functions become available on the top two rows of the keyboard. A minimum size required for FI is SIZE 010.



The functions provided are summarized below.

KEY	FUNCTION	(FLAG/REG)
a	Multiplies contents of X by 12 and stores result as n. n=12X	R01
b	Divides contents of X by 12 and stores result as %i. %i=X/12	R02
c	Toggles flag 08 to specify Continuous (F08 set) or Discrete (F08 clear) compounding. Status display shows C or D.	F08
d	Toggles flag 09 to specify Beginning of period payments (F09 set) or End of period payments (F09 clear). Status display shows B or E.	F09
e	Clears financial registers n, %i, PV, PMT, FV, and sets the compounding frequency (R08) and the payment frequency (R09) both to the default value of 1. CF=PF=1. Also displays a two character mnemonic indicator of the status of F08 and F09.	R01-R05 R08 R09

A	Enters or Solves for number of periods. n is the total number of payments during the full term of the transaction, or if no payments are made n is the total number of compounding periods.	R01 n
B	Enters or Solves for the interest rate.** %i is the nominal rate for the period implied by the compounding and payment frequency values in R08 and R09 (usually the nominal annual rate).	R02 %i
C	Enters or Solves for the present value. Use standard financial sign conventions.	R03 PV
D	Enters or Solves for periodic payment. Use standard financial sign conventions.	R04 PMT
E	Enters or Solves for future value. Use standard financial sign conventions.	R05 FV
H	Enters the compounding frequency. CF is the number of times the interest rate is compounded during the period implied by the interest rate %i. When continuous compounding is specified the value in R08 is ignored.	R08 CF
I	Enters the payment frequency. PF is the number of payment periods occuring during the period implied by the interest rate %i. When no payments are involved PF must be set equal to CF. For continuous compounding cases where PMT = 0, set PF = 1.	R09 PF
J	Displays a two character mnemonic indicator of the status of flags F08 and F09. C = Continuous D = Discrete E = End of period B = Beginning of period	

** When solving for %i the first guess and succeeding approximations of i (decimal) may be VIEWed during each iteration by setting flag 10. F10

WARNING: Solutions using or resulting in a zero rate of interest (%i) will cause a "DATA ERROR".

MORE EXAMPLES OF FI

In the keystroke solutions shown for each example, the lower case letters a through e represent shifted key functions of keys A through E. Key in the indicated quantities and press the user defined keys as indicated in the "Do" column. Contents of the display at significant points in the solution are shown in the "See" column and are followed by identification in the "Result" column. Before running these examples, perform "MEMORY LOST" and set FIX 2 display mode. A suggestion is to go through all 15 examples, one after the other. Key GTO " FI " and set USER mode.

Example 1: Simple Interest. Find the annual simple interest rate (%) for an $800 loan to be repayed at the end of one year with a single payment of $896.

Do:	See:	Result:
e	"DE"	Clear, Discrete/End status
1 A	1.00	n=1
800 CHS C	-800.00	PV=$800.00
896 E	896.00	FV=$896.00
B	12.00	APR=%i=12%

Example 2: Compound Interest. Find the future value of $800 after one year at a nominal rate of 12% compounded monthly. No payments are specified, so the payment frequency is set equal to the compounding frequency.

Do:	See:	Result:
e	"DE"	Clear, Discrete/End status
12 A	12.00	n=12
H I	12.00	CF=PF=12
12 B	12.00	keyboard input required to store NAR=12%=%i
800 CHS C	-800.00	PV=$800.00
E	901.46	FV=$901.46

Example 3: Periodic Payment. Find the monthly end-of-period payment required to fully amortize the loan in Example 2. A fully amortized loan has a future value of zero. Use data retained from Example 2.

Do:	See:	Result:
0 E	0.00	Set FV=$0.00
D	71.08	PMT=$71.08

Example 4: Conventional Mortgage. Find the number of monthly payments necessary to fully amortize a loan of $100,000 at a nominal rate of 13.25% compounded monthly, if end-of-period payments of $1,125.75 are made.

Do:	See:	Result:
e	"DE"	Clear, Discrete/End status
12 H I	12.00	CF=PF=12
13.25 B	13.25	NAR=13.25%=%i
100000 C	100,000.00	PV=$100,000.00
1125.75 CHS D	-1,125.75	PMT=$1,125.75
A	360.10	#pmts=n=360.10

Example 5: Final Payment. Using the same data as in the preceding example, find the amount of the final payment if n is changed to 360. The final payment will be equal to the regular payment plus any balance (FV) remaining at the end of period number 360.

Do:	See:	Result:
360 A	360.00	Set n=360 exactly
E	-108.87	FV=$108.87
RCL 04	-1,125.75	Recall PMT
+	-1,234.62	Final PMT=$1,234.62

Example 6: Balloon Payment. On long term loans, small changes in the periodic payments can generate large changes in the future value. If the monthly payment in the preceding example is rounded down to $1,125.00 how much additional (balloon) payment will be due with the final regular payment?

Do:	See:	Result:
1125 CHS D	-1,125.00	Set PMT=$1,125.00 even
E	-3,580.00	Additional balloon payment = $3,580.00

Example 7: Canadian Mortgage. Find the monthly end-of-period payment necessary to fully amortize a 25 year $85,000 loan at 11% compounded semiannually.

Do:	See:	Result:
e	"DE"	Clear, Discrete/End status
2 H	2.00	CF=2
12 I	12.00	PF=12
25 a	300.00	n=300
11 B	11.00	NAR=11%=I%
85000 C	85,000.00	PV=$85,000.00
D	-818.15	PMT=$818.15

Example 8: European Mortgage. The "effective annual rate (EAR) is used in some countries (especially in Europe) in lieu of the nominal annual rate commonly used in the United States and Canada. For a 30 year $90,000 mortgage at 14% (EAR) compute the monthly end-of-period payments. When using an EAR, the compounding frequency (CF) is set to 1.

Do:	See:	Result:
e	"DE"	Clear, Discrete/End status (CF=1 after clearing)
12 I	12.00	PF=12
30 a	360.00	n=360
14 B	14.00	EAR=14%=%I
90000 C	90,000.00	PV=$90,000.00
D	-1,007.88	PMT=$1,007.88

Example 9: Bi-Weekly Savings. Compute the future value of bi-weekly savings of $100 for 3 years at a nominal annual rate of 5.5% compounded daily. Note: Set status to "DB".

Do:	See:	Result:
e	"DE"	Clear, Discrete/End status
d	"DB"	Discrete/Begin Status
365 H	365.00	CF=365
26 I	26.00	PF=26
3 X A	78.00	n=3x26=78
5.5 B	5.50	NAR=5.5%=%I
100 CHS D	-100.00	PMT=$100.00
E	8,489.32	FV=$8,489.32

Example 10: Present Value - Annuity Due. What is the present value of $500 to be received at the beginning of each quarter over a 10 year period if money is being discounted at 10% NAR compounded monthly?

Do:	See:	Result:
e	"DB"	Clear, Discrete/Begin status
12 H	12.00	CF=12
4 I	4.00	PF=4
40 A	40.00	n=40
10 B	10.00	NAR=10%=%I
500 D	500.00	PMT=$500.00
C	-12,822.64	PV=$12,822.64

Example 11: Balloon Payment @ n+1. Compute the monthly end-of-period payment on a 3 year $20,000 loan at 15% NAR compounded monthly, with a $10,000 balloon payment due at the end of the 37th period. The balloon payment must be discounted one period to make it coincide with the last regular payment. Note: Set status to "DE".

Do:	See:	Result:
e	"DB"	Clear Financial
d	"DE"	Set Discrete/End status
12 H I	12.00	CF=PF=12
3 a	36.00	n=36
15 B	15.00	NAR=15%=%I
20000 C	20,000.00	PV=$20,000.00
XEQ 07	0.00	Calculate I as a decimal and leave in R07
10000	10,000.00	Start calculation of 10,000/(1+I) = FV
RCL 07 1 +	1.01	1+I
/ CHS E	-9876.54	FV=$9,876.54 (discounted)
D	-474.39	PMT=$474.39

The balloon payment was discounted by executing LBL 07 (XEQ 07) to develop the effective interest rate in R07. The $10,000 was then divided by (1+I) and entered in E as the discounted future value of the balloon payment.

Example 12: Effective Rate - 365/360 Basis. Compute the effective annual rate (%APR) for a nominal annual rate of 12% compounded on a 365/360 basis used by some Savings & Loan Associations.

Do:	See:	Result:
FIX 3 e	"DE"	Set up display & status
365 A	365.000	n=365
H	365.000	CF=365
360 I	360.000	PF=360
12 B	12.000	NAR=12%=%I
100 CHS C	-100.000	PV=$100.00
E	112.935	FV=$112.94
RCL 03 +	12.935	%APR=12.935%
FIX 2	12.94	Return to normal display

Example 13: Mortgage with "points". What is the true APR of a 30 year, $75,000 loan at a nominal rate of 13.25% compounded monthly, with monthly end-of-period payments of $844.33 if 3 "points" are charged? The PV must be reduced by the dollar value of the points and/or any lenders fees to establish an effective PV. Because the payments remain the same the true APR will be higher than the nominal rate.

Do:	See:	Result:
e	"DE"	Clear, Discrete/End status
12 H I	12.00	CF=PF=12
30 a	360.00	n=360
75000 ENTER↑		
3 XEQ "%" - C	72,750.00	PV=$72,750.00
844.33 CHS D	-844.33	PMT=$844.33
B	13.69	True APR=13.69%

Example 14: Equivalent Payments. Find the equivalent monthly payment required to amortize a 20 year $40,000 loan at 10.5% NAR compounded monthly, with 10 annual payments of $5,029.71 remaining. Compute PV of the remaining annual payments, then change n and PF to a

monthly basis and compute the equivalent monthly PMT.

Do:	See:	Result:
e	"DE"	Clear, Discrete/End status (PF=1 after clearing)
12 H	12.00	CF=12
10 A	10.00	n=10
10.5 B	10.50	NAR=10.5%=%I
5029.71 CHS D	-5,029.71	PMT=$5,029.71
C	29,595.88	PV=$29,595.88
12 I	12.00	PF=12, set monthly basis
10 a	120.00	n=120 (monthly)
D	-399.35	PMT=$399.35 (monthly)

Example 15: Perpetuity - Continuous Compounding
If you can purchase a single payment annuity with an initial investment of $60,000 that will be invested at 15% NAR compounded continuously, what is the maximum monthly return you can receive without reducing the $60,000 principal? If the interest rate is constant and the principal is not disturbed the payments can go on indefinitely (a perpetuity). Note that the term "n" of a perpetuity is immaterial. It can be any non-zero value. Set status to "CE".

Do:	See:	Result:
e	"DE"	Clear, Discrete/End status (CF=1 after clearing)
c	"CE"	Continuous/End status
12 A	12.00	n=12
I	12.00	PF=12
15 B	15.00	NAR=15%=%I
60000 E	60,000.00	FV=$60,000.00
CHS 1 X C	-60,000.00	Data entry flag is set so PV is stored as $60,000.00
D	754.71	PMT=$754.71

SUPPORTIVE PROGRAMS FOR FI

There are two optional routines provided below to extend the capability of the ROM routine FI. These routines are not located in the ROM, and must be loaded into RAM memory for their execution. They are named LPAS and FAST.

1. LBL LPAS

LBL LPAS "Loan Payments and Amortization Schedule" is really a full program in its own right, although it does use ROM routines FI, CJ, and CP. LPAS extends the capabilities of FI to accommodate "shifted" payment situations, when the first periodic payment does not fall at the beginning (BEGIN) or the end (END) of the first period, but at any date after the effective date. LPAS also provides an amortization schedule as an option.

2. LBL FAST - Reducing Interest Solution Time

LBL FAST is an optional routine used when solving for interest. Its purpose is to provide an initial starting guess for the interest-solving loop which is closer to the exact solution than that provided by LBL FI initial guess. The result is that interest solving execution time is usually shorter.

Don Dewey (5148) produced both supporting programs.

APPLICATION PROGRAM 1 FOR FI

LPAS - Loan Payments and Amortization Schedule

The FI program, like most financial programs and calculators, assumes that the first periodic payment occurs on either the first or last day of the payment period as specified by the beginning of period/end of period switch or toggle. Many financial agreements do not follow this convention. An agreement may call for the regular periodic payments to start earlier or later in order to provide a better match to other cash flow considerations of the borrower or lender. These agreements with "shifted" initial payment dates can be handled by conventional financial programs by computing an effective present value (PV) that compensates for the difference in interest accrued during the irregular first payment period. This computation becomes more complex when the compounding and payment frequencies (CF and PF) are unequal.

Shifting the initial payment date forces a change in the number or amount of the periodic payments or in the amount of the final or balloon payment. However, the participants to an agreement may want to specify the number and/or amount of the regular payments, and adjust the final payment to complete the amortization. Even without a shifted initial payment date or other restrictions the regular periodic payments seldom precisely complete the amortization and the final payment must be adjusted to accomplish this.

For the uninitiated or infrequent user of financial programs, the accomodation of a shifted first payment date and/or the computation of the correct final payment amount can cause problems. The following program easily handles these cases and also takes the drudgery out of computing an amortization schedule.

The LPAS program uses the FI program and the CJ and CP routines in the PPC ROM to expand the capabilities of the FI program to accomodate "shifted" initial payment dates and to compute the number and amount of periodic payments, and the final payment required to amortize a loan or to accumulate a specific future value. The information needed to prepare a loan amortization schedule may also be computed on an optional basis. The extensive capabilities of the FI program are used in their normal manner to define the parameters of a specific problem and to develop the initial solution. Two additional input parameters are provided; the effective date (ED) and the initial payment date (IP). These two dates define the length of the first payment period which need not be equal to the normal payment period implied by the payment frequency value (PF). The initial payment date (IP) also establishes the number of payments that will occur in the first year. The program computes the regular periodic payment and the final payment required to amortize a loan or to accumulate a specified future value over a specified term (n), or the number of payments and the final payment necessary to amortize a loan or to accumulate a specified future value with a specified periodic payment amount.

Conventional loans, mortgages with or without balloon payments, and Canadian or European mortgages are all acceptable to the LPAS program. Cases with payment frequencies of semi-monthly (PF=24) or less, use a 30 day month convention for determining the number of days of shift in the first payment date and the number of payments occurring in the first year. For payment frequencies greater than semi- monthly (i.e., daily, weekly, or bi-weekly) the acutal number of calendar days is used.

LPAS Program - Operation

The LPAS program computes the regular periodic payment and the final payment for both present value (PV*) and future value (FV*) cases. PV* cases involve periodic payments that reduce or amortize a present value. FV* cases involve appreciation or accumulation to a future value. The amortization schedule portion of the program supports PV* cases only. The LPAS program can be used with or without a printer (CF21).

The FI program is accessed and used to set the status (CB, CE, DB, DE), the compounding frequency (CF), the payment frequency (PF), the standard financial values (n, %i, PV, PMT, FV) and to solve for any missing financial value. Note: The FI program can be accessed by pressing "J" when the LPAS program has control. After entering the normal financial program data, the effective date of the financial agreement (ED) and the date of the initial payment (IP) are entered into the X and Y registers in the form MM.DDYYYY (Y=ED, X=IP). The IP date must not be earlier than the ED date.

The LPAS program is then executed. For easy access the LPAS program should be assigned to a key. The LPAS program was assigned to the X<>Y (F) key in the keystroke solutions in the example programs below. The program computes the regular periodic payment required to maintain the specified interest rate. The computation compensates for any fractional portion of the term and for any deviation from the normal initial payment date. When the program first stops, the computed payment (rounded to two decimal places) is in the PMT register (R04) and is displayed in the X register.

First Stop - The computed payment may be accepted, or a modified payment may be entered and substituted by pressing key "D". To continue the computation, select one of the following two options:

1. By pressing "H" the amortization period is limited to the integer portion of the term (n) and the final or balloon payment is adjusted to complete the amortization.

2. By pressing "J" the term (n) is recomputed to accomplish the amortization with the specified periodic payment with a minimum adjustment to the final or balloon payment.
The amortization choice restarts the program and the number of periodic payments and the amount of the final payment are computed. At the second stop the stack contains:

T = number of payments occuring in first year
Z = number of regular periodic payments
Y = amount of the regular periodic payment
X = amount of the combined final and balloon payments

Second Stop - An amortization schedule may be computed by pressing "E" (for PV* cases only) or control may be returned to the FI program by pressing "J".

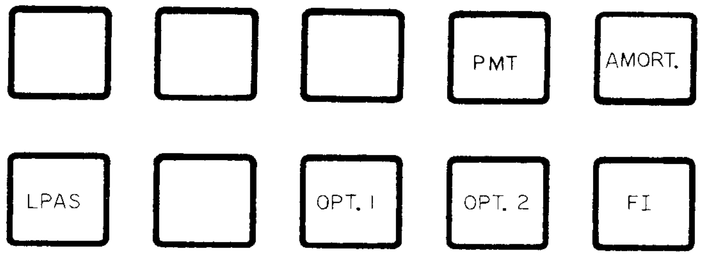
If an amortization schedule is computed and a printer is not available (FC?21) the program will stop after computing the values for each year. At each stop the stack will contain:

T = cumulative interest paid
Z = balance outstanding after last PMT for year
Y = interest paid during the year
X = year (YY)

To compute the amortization data for each succeeding year press R/S. Completion of the amortization is indicated by ** in the display. The total interest paid is in the Y register at this final stop.

After completion of the amortization, control may be returned to the FI program by pressing "J".

Keyboard Functions: (LPAS Program)



Key	Function	(Flag/Reg)
D	Enter revised periodic payment "PMT"	(R04)
E	Compute amortization data (PV* case only)	(R00-R13)
F	Enter ED and IP dates and compute periodic PMT	
H	Select Option 1 and compute final PMT	(F07)
I	Select Option 2 and compute term n and final PMT	
J	Transfer to FI program and display status	
R/S	Compute amortization data for next year	

Program requirements and limitations. CJ, CP, FI are the PPC ROM required routines. LPAS is 655 bytes SIZE=014 Flags 06-10,21,28,29

Acceptable Payment Frequencies (PF) are:

1 = annual	12 = monthly
2 = semi-annual	24 = semi-monthly
3 = tri-annual	26 = bi-weekly
4 = quarterly	52 = weekly
6 = bi-monthly	365 = daily

WARNING: Solutions using or resulting in a zero rate of interest (%i) will cause a "DATA ERROR".

The output of LPAS is printed in three sections separated by horizontal lines. The first section records the original parameters of the case. The second section records the amount and number of regular payments and the final payment necessary to satisfy the options selected. The third section displays the optional amortization schedule.

Examples:

In the keystroke solution for each example, the lower case letters a through e represent shifted functions of keys A through E. Key in the indicated quantities and press the user defined keys as indicated in the "Do" column. Contents of the display or the printed output at significant points in the solution are shown in the "See" column and are followed by identification in the "Result" column. Use FIX 2 display mode, and assign LPAS to key F (X<>Y).

Example A: Conventional Mortgage. Develop the data for an amortization schedule for a fully amortized 30-year, $100,000 mortgage at 14.75% NAR compounded monthly with end of period payments of $1,244.48 with the first payment due on November 1, 1981. Effective date of the loan is September 25, 1981. Use option 1 (H) to limit the amortization period to 360 payments.

Do:	See:	Result:
CLX STO 06	0.00	Store FI function call
XEQ " FI "	"DE"	Discrete/End status
12 H I	12.00	CF=PF=12
30 a	360.00	n=360
14.75 B	14.75	NAR=14.75%=%I
100000 CHS C	-100,000.00	PV=$100,000.00
1244.48 D	1,244.48	PMT=$1,244.48
E	-27.98	FV=$27.98
9.251981 ENTER↑		
11.011981 F	1,247.52	PMT, shifted IP $1,247.52
H	1,248.31	Final PMT=$1.248.31
E	**	Compute amortization data, see print out below.

```
EXAMPLE A   CF=12  PF=12
************************
PV*DE   PV   -100,000.00
  360 PMTS      1,244.48
14.750% FV        -27.98
ED  9-25-81 IP 11- 1-81*
xxxxxxxxxxxxxxxxxxxxxxxx
  359 PMTS      1,247.52
+FINAL PMT      1,248.31
xxxxxxxxxxxxxxxxxxxxxxxx
YR  INTEREST  ENDING BAL
81     2,464.   100,214.
82    14,768.   100,012.
83    14,736.    99,778.
84    14,699.    99,507.
85    14,657.    99,193.
86    14,607.    98,830.
87    14,550.    98,410.
88    14,483.    97,923.
89    14,407.    97,359.
90    14,318.    96,707.
```

```
91    14,214.    95,951.
92    14,095.    95,076.
93    13,957.    94,063.
94    13,797.    92,889.
95    13,612.    91,531.
96    13,397.    89,958.
97    13,149.    88,137.
98    12,861.    86,028.
99    12,528.    83,586.
00     2,143.    80,758.
01    11,696.    77,485.
02    11,179.    73,694.
03    10,581.    69,305.
04     9,888.    64,222.
05     9,085.    58,338.
06     8,156.    51,524.
07     7,080.    43,634.
08     5,835.    34,498.
09     4,392.    23,920.
10     2,722.    11,672.
11       804.         0.
**   348,860.
```

Note: 2 payments in 1981. The negative amortization during the first two years is due to the delayed first payment date. The asterisk following the IP date indicates a shifted initial payment date.

If a printer is not used when working Example A, after execution the stack will contain the following:

after F	after H	after E*
T= -	T= 2.00	T = 2,464. Σ Int.
Z= -	Z= 359.00	Z = 100,214. E.Bal.
Y= -	Y= 1,247.52	Y= 2,464. Yr.Int.
X= 1,247.52	X= 1,248.31	X= 81. Year

*Press R/S to advance amortization to next year. the end of the amortization is indicated by ** in display.

Example B: Sinking Fund / Savings Plan Starting with an initial deposit of $3,000 compute the number of bi-weekly deposits of $200 and the amount of the final deposit needed to accumulate a balance of $20,000 in an account paying 8% compounded continuously, if the intial deposit (PV) is made on December 1, 1981 and the first bi-weekly deposit (PMT) is made on December 11, 1981. Set the status to CB.

Do:	See:	Result:
J	"DE"	Return to FI
c	"CE"	Set Continuous compounding
d	"CB"	Set Beginning of period payments
e	"CB"	Clear Financial Status=Continuous/Beginning CF=1 after clearing
26 I	26.00	PF=26
8 B	8.00	NAR=8%=%I
3000 CHS C	-3,000.00	PV=$3,000.00
200 CHS D	-200.00	PMT=$200.00
20000 E	20,000.00	FV=$20,000.00
A	72.43	n=72.43
12.011981 ENTER↑		
12.111981 F	-197.29	PMT, shifted IP $197.29
200 CHS D	-200.00	Enter revised PMT $200.00
I	-91.67	Final PMT=$91.67
E	"FV* ?"	Indicates attempted amortization of FV* case

```
EXAMPLE B  CF=1 PF=26
************************
FV*CB  PV      -3,000.00
  72+ PMTS       -200.00
 8.000% FV     20,000.00
ED 12- 1-81 IP 12-11-81*
xxxxxxxxxxxxxxxxxxxxxxxx
  71 PMTS        -200.00
+FINAL PMT        -91.67
```

FV*CB = Future Value case with Continuous compounding and Beginning of period payments/deposits. The plus (+) sign following the number of payments indicates that the term includes a fractional payment period as developed from the original specifications.

Example C: Loan with Balloon Payment. Develop the amortization data for a $500,000 loan at 15% NAR with monthly compounding, to be repaid with 30 monthly end of period payments of $20,000 and a balloon payment of $3,225.30 coincident with the final payment. The loan effective date is September 14, 1981 and the first payment is scheduled for October 14, 1981.

Do:	See:	Result:
J	"CB"	Return to FI Status from previous example
c	"DB"	Set Discrete compounding
d	"DE"	Set End of period payments
e	"DE"	Clear Financial, final status= Discrete/End
12 H I	12.00	CF=PF=12
30 A	30.00	n=30
15 B	15.00	NAR=15%=%I
500000 CHS C	-500,000.00	PV=$500,000.00
20000 D	20,000.00	PMT=$20,000.00
E	3,225.30	Balloon=$3,225.30
9.141981 ENTER↑		

10.141981 F	20,000.00	PMT=$20,000.00
H	23,225.30	Final + Balloon = $23,225.30
E	**	Compute amortization data, see print out below.

```
EXAMPLE C  CF=12 PF=12
++++++++++++++++++++++++
PV*DE   PV   -500,000.00
   30 PMTS    20,000.00
15.000% FV     3,225.30
ED  9-14-81 IP 10-14-81
xxxxxxxxxxxxxxxxxxxxxxxx
   29 PMTS    20,000.00
+FINAL PMT    23,225.30
xxxxxxxxxxxxxxxxxxxxxxxx
YR  INTEREST  ENDING BAL
81    18,232.   458,232.
82    56,456.   274,688.
83    26,950.    61,638.
84     1,587.         0.
**   103,225.
```

Because the initial payment occurs exactly one month after the loan effective date there is no change in the re-computed PMT.

Example D: Delayed First Payment This example will illustrate the effect of a different repayment plan for the loan defined in Example C. Develop the data for amortizing a $500,000 loan at 15% NAR with monthly compounding, to be repaid with 60 semi-monthly end of period payments of $10,000 and a balloon payment coincident with the final payment. The loan effective date is September 14, 1981 and the first payment is scheduled for November 1, 1981.

Do:	See:	Result:
J	"DE"	Return to F1 Status left from Example C
12 H	12.00	CF=12
24 I	24.00	PF=24
60 A	60.00	n=60
15 B	15.00	NAR=15%=%I
500000 CHS C	-500,000.00	PV=$500,000.00
10000 D	10,000.00	PMT=$10,000
E	974.25	FV=$974.25
9.141981 ENTER↑		
11.011981 F	10,268.92	PMT=$10,268.92
10000 D	10,000.00	Set PMT=$10,000.00 exactly
H	30,466.27	Final+Balloon=$30,466.27
E	**	Compute amortization data See print out below

```
EXAMPLE D  CF=12 PF=24
++++++++++++++++++++++++
PV*DE   PV   -500,000.00
   60 PMTS    10,000.00
15.000% FV       974.25
ED  9-14-81 IP 11- 1-81*
xxxxxxxxxxxxxxxxxxxxxxxx
   59 PMTS    10,000.00
+FINAL PMT    30,466.27
xxxxxxxxxxxxxxxxxxxxxxxx
YR  INTEREST  ENDING BAL
81    12,541.   485,968.
82    60,113.   306,081.
83    31,195.    97,277.
84     3,189.         0.
**   107,038.
```

The total interest on this repayment plan is $3,813 more than in Example C due to the delayed first payment date and the smaller payments. The borrower has the use of more money for a longer time.

LPAS Program - Equations

All equations assume the use of standard financial transaction sign conventions of money received as positive (+) and money paid out as negative (-).

Notation used:

- d = number of days in payment period
- i_e = effective interest rate per payment period
- m = integer portion of term n
- n = number of payment periods in term
- s = number of days first payment is shifted
- CF = compounding frequency per year
- ED# = effective date - day number
- FV = future value after n periods
- FV_m = future value after m periods
- FV_{m-1} = future value after m-1 periods
- FV* = future value case
- INT = interest for the year
- IP# = initial payment date - day number
- NP = number of payments in the year
- PF = payment frequency per year
- PMT = periodic payment
- PMT_f = final payment
- PV = present value
- PV_e = effective present value
- PV* = present value case

If $|FV| <= |PV|$, then PV* case
If $|FV| > |PV|$, then FV* case

The initial payment date is "shifted" when: $s \neq 0$

where: $s = IP\# - ED\#$ for beginning of period payments

$s = IP\# - ED\# + d$ for end of period payments

For financial calculations involving a "shifted" first payment date, the present value (PV) must be converted to an effective present value (PV_e) that is adjusted to compensate for the difference in interest accrued during the irregular first payment period.

$$PV_e = PV(1+i_e)^{(sPF/dCF)}$$

To precisely complete the amortization of a present value or the accrual of a future value, the final payment must be calculated separately from the regular periodic payment. The LPAS program incorporates eight variations of final payment calculations.

$PMT_f = FV_{m-1}$	PV* case, annuity due, Option 1
$= FV_m$	PV* case, annuity due, Option 2
$= FV_m + PMT$	PV* case, ordinary annuity, Option 1

$PMT_f = FV_m(1+I_e)$ PV* case, ordinary annuity Option 2

$= FV_{m-1} - FV/(1+I_e)$ FV* case, annuity due, Option 1

$= FV_m - FV/(1+I_e)$ FV* case, annuity due Option 2

$= FV_m + PMT - FV$ FV* case, ordinary annuity Option 1

$= FV_m(1+I_e) - FV$ FV* case, ordinary annuity Option 2

N.B. Values m and n are different for Options 1 and 2

The interest paid during each year of amortization is determined by the difference between the ending and beginning balances plus the sum of the payments for the year.

INT = (NP*PMT) + PV + FV

LPAS Program - Line by Line Analysis

LBL LPAS - Mainline - First Section

001	store ED and IP dates.	Print separator line	
007	set flag F06 (FV* case) if: |FV|>|PV|		
014	calculate term (n).	Print line 1	(PV)
026	Format data.	Print line 2	(n)(PMT)
038	Format date.	Print line 3	(%I)(FV)
044	If PF>24 set Flag 07 (calendar year basis)		
050	Calculate day number of effective date		(ED#)
053	Calculate day number of first PMT date		(IP#)
056	Develop number of days from ED thru IP date		(s)
058	Develop number of day from IP thru year end		
069	Develop number of days in normal PMT period		(d)
079	Adjust s for end of period payments		(s)
081	Develop number of payments in first year		
086	If PMT = 0, set s = 0		
090	If s ≠ 0, append *.	Print line 4	(ED)(IP)
095	Develop PV_e to adjust for shifted IP date		(PV_e)
107	Save IP year (YY) and calculate payment		(PMT)
113	--FIRST STOP--		

At this stop the calculated periodic payment may be accepted or the original or a modified payment can be entered and stored by pressing key D before selecting an amortization option (H or I).

Subroutines

LBL 01 Reformat date for CJ and load print buffer
LBL 02 Calculate day number using 30/360 convention
LBL 03 Calculate day number using CJ (calendar basis)
LBL 04 Format month (MM) and day (DD) for printing
LBL 05 Display control -
06 - and column format subroutine
LBL 07 Execute specified FI routine
LBL 08 Fill buffer with specified character -
09 - and printer separator line
LBL D Store PMT in R04
LBL J Transfer control to FI and display status

- Mainline - Second Section

LBL H Option 1 - If PMT≠0, set flag F07 (set=opt. 1)
LBL I Option 2
205 Calculate new (n). If n=0 use original n (n)
213 Select (n): option 1=original option 2=new

216 Calculate FV and modify to -
LBL 10 - develop final payment (PMT_f)

LBL 11 Store final PMT_f Print separator line
263 Format data. Print line 5 (n-1)(PMT)
269 Format data. Print line 6 (PMT_f)
278 --SECOND STOP--

At this stop the amortization schedule calculation may be selected by pressing key E (for PV* cases only), or control may be returned to the FI program via key J.

Subroutine

LBL 12 Format control subroutine

LBL E - Mainline - Third Section - Amortization
284 If FV* case stop and display "FV* ?" (invalid)
288 Print separator line. Print heading line
294 Reduce payment count by number 1st yr payments
LBL 13 Develop interest for year -
14 - and calculate ending balance
LBL 15 ΣINT and format data. Print amortization line
346 Load stack for review and stop if FC?21
351 --AMORTIZATION YEAR STOP--

If flag F21 is cleared this stop will occur after the amortization calculations have been made for each year. Amortization data is available in the stack.

352 If not final year, update year & payment count
LBL 16 End routine Print total (**)(ΣINT)
384 END

Other LPAS program technical details:

Global Label: LPAS
Local Labels: D,E,H,I,J, and 01-16
Byte Count: 655 (requires one memory module)
Size Required: SIZE=014
ROM Routines called: CJ, CP, FI
Subroutine Levels: 3
Flags Used: LPAS - 06,07,21,28,29
FI - 08, 09, & 10
CJ - 10
CP - 29 & 40

Data Registers Used:

R00: multi use store	R07: i as decimal
R01: n term	R08: CF compounding freq.
R02: %i as percentage	R09: PF payment frequency
R03: PV present value	R10: multi use store
R04: PMT periodic pmt	R11: multi use store
R05: FV future value	R12: multi use store
R06: IND addr.	R13: multi use store

Status Registers: none used
Alpha Registers: all used
Σ REG: not used
Peripherals: printer recommended but not required
Stack Usage: I/O see program description
Execution Time: variable

BAR CODE ON PAGE 480

APPLICATION PROGRAM FOR: FI

```
01♦LBL "LPAS"
02 STO 10
03 X<>Y
04 STO 00
05 0
06 XEQ 08
07 CF 06
08 RCL 03
09 ABS
10 RCL 05
11 ABS
12 X>Y?
13 SF 06
14 1
15 XEQ 07
16 ASTO X
17 "P"
18 FS? 06
19 "F"
20 "⊢V*"
21 ARCL X
22 "⊢  PV"
23 RCL 03
24 XEQ 05
25 ADV
26 XEQ 12
27 RCL 01
28 ENTER↑
29 INT
30 ARCL X
31 -
32 X≠0?
33 "⊢+"
34 "⊢ PMTS"
35 RCL 04
36 XEQ 05
37 ADV
38 FIX 3
39 ARCL 02
40 "⊢% FV"
41 RCL 05
42 XEQ 05
43 ADV
44 XEQ 12
45 CF 07
46 24
47 RCL 09
48 X>Y?
49 SF 07
50 "ED "
51 RCL 00
52 XEQ 01
53 X<> 10
54 "⊢ IP "
55 XEQ 01
56 ST- 10
57 STO 00
58 FIX 2
59 SF 28
60 SF 29
61 1
62 ST+ 11
63 STO 12
64 CLX
65 STO 13
66 XEQ 02
67 RCL 00
68 -
69 360
70 ENTER↑
71 6
72 FC?C 07
73 CLX
74 +
75 RCL 09
76 /
77 INT
78 STO 13
79 FC? 09
80 ST+ 10
81 +
82 LASTX
83 /
84 INT
85 STO 12
86 RCL 10
87 RCL 04
88 X≠0?
89 X<>Y
90 CHS
91 X≠0?
92 "⊢*"
93 FS? 21
94 PRA
95 CLA
96 RCL 08
97 RCL 13
98 *
99 /
100 RCL 09
101 *
102 RCL 07
103 LN1+X
104 *
105 E↑X
106 ST* 03
107 RCL 06
108 STO 11
109 4
110 XEQ 07
111 RND
112 STO 04
113 RTN
114♦LBL 01
115 INT
116 -100
117 STO 11
118 STO Z
119 X<>Y
120 STO 12
121 XEQ 04
122 INT
123 STO 13
124 XEQ 04
125 CHS
126 ST* 11
127 FRC
128 *
129 STO 06
130 10
131 X>Y?
132 "⊢0"
133 ARCL Y
134♦LBL 02
135 FS? 07
136 GTO 03
137 RCL 11
138 360
139 *
140 RCL 12
141 30
142 *
143 +
144 RCL 13
145 +
146 RTN
```

APPLICATION PROGRAM FOR: FI

```
147♦LBL 03
148 RCL 11
149 RCL 12
150 RCL 13
151 XROM "CJ"
152 RTN
153♦LBL 04
154 10
155 X>Y?
156 "⊢ "
157 ARCL Y
158 RDN
159 LASTX
160 -
161 *
162 "⊢-"
163 RTN
164♦LBL 05
165 FIX 2
166 9
167♦LBL 06
168 STO 06
169 X<>Y
170 SF 28
171 SF 29
172 FC? 21
173 RTN
174 ACA
175 XROM "CP"
176 CLA
177 RTN
178♦LBL 07
179 STO 06
180 XROM "FI"
181 RTN
182♦LBL 08
183 FC? 21
184 RTN
185 24
186 X<>Y
187♦LBL 09
188 ACCHR
189 DSE Y
190 GTO 09
191 PRBUF
192 RTN
193♦LBL D
194 STO 04
195 RTN
196♦LBL J
197 10
198 STO 06
199 GTO "FI"
200♦LBL H
201 RCL 04
202 X≠0?
203 SF 07
204♦LBL I
205 RCL 01
206 INT
207 STO 10
208 1
209 XEQ 07
210 INT
211 X=0?
212 RCL 10
213 FC? 07
214 STO 10
215 RCL 10
216 1
217 FS? 07
218 ST- 10
219 FS? 07
220 FC? 09
221 CLX
222 -
223 STO 01
224 RCL 07
225 1
226 +
227 STO 13
228 RCL 05
229 STO 00
230 5
231 XEQ 07
232 RCL 00
233 STO 05
234 FC? 06
235 CLX
236 STO 00
237 X<>Y
238 RCL 13
239 FS? 09
240 ST/ 00
241 X<>Y
242 FC? 09
243 GTO 10
244 RCL 00
245 -
246 GTO 11
247♦LBL 10
248 FC? 07
249 *
250 RCL 00
251 FC? 06
252 CLX
253 -
254 RCL 04
255 FC? 07
256 CLX
257 +
258♦LBL 11
259 RND
260 STO 13
261 1
262 XEQ 08
263 XEQ 12
264 ARCL 10
265 "⊢ PMTS"
266 RCL 04
267 XEQ 05
268 ADV
269 "+FINAL PMT"
270 RCL 13
271 XEQ 05
272 ADV
273 RCL 12
274 RCL 10
275 RCL 04
276 RCL 13
277 FIX 2
278 RTN
279♦LBL 12
280 FIX 0
281 CF 28
282 CF 29
283 RTN
284♦LBL E
285 "FV* ?"
286 FS? 06
287 PROMPT
288 1
289 XEQ 08
290 "YR  INTEREST "
291 "⊢ENDING BAL"
292 FS? 21
```

Listing continued on page 158.

Listing continued from 157.

BAR CODE ON PAGE 480

APPLICATION PROGRAM FOR: FI

```
293 PRA
294 CF 07
295 CLA
296 CLX
297 X<> 12
298 RCL 10
299 X<=Y?
300 SF 07
301 X<>Y
302 STO 01
303 -
304 STO 10
305*LBL 13
306 XEQ 12
307 RCL 11
308 10
309 X>Y?
310 "⊢0"
311 ARCL Y
312 RCL 03
313 RCL 04
314 RCL 01
315 *
316 +
317 STO 00
318 CLX
319 STO 05
320 FC? 07
321 GTO 14
322 RCL 13
323 ST+ 00
324 GTO 15
325*LBL 14
326 5
327 XEQ 07
328 FIX 2
329 RND
330 ST+ 00
331 STO 05
332 CHS
333 STO 03
334*LBL 15
335 FIX 0
336 RCL 00
337 RND
338 ST+ 12
339 8
340 XEQ 06
341 RCL 05
342 RND
343 8
344 XEQ 06
345 ADV
346 RCL 12
347 RCL 05
348 RCL 00
349 RCL 11
350 FC? 21
351 STOP
352 FS?C 07
353 GTO 16
354 1 E2
355 RCL 11
356 1
357 +
358 X=Y?
359 -
360 STO 11
361 RCL 10
362 STO 01
363 RCL 09
364 ST- 10
365 X<=Y?
366 STO 01
367 -
368 X<=0?
369 SF 07
370 GTO 13
371*LBL 16
372 "**"
373 RCL 12
374 8
375 XEQ 06
376 11
377 FS? 21
378 SKPCHR
379 ADV
380 "**"
381 ASTO X
382 FIX 2
383 .END.
```

APPLICATION PROGRAM 2 FOR FI

FAST - Reducing Interest Solution Time

When the solution for interest is required for PMT≠0, LBL 02 of FI produces an initial guess for the interest which is supplied to the iterative loop starting at LBL 06. In most cases the LBL 02 guess is usually "close" (in the mathematical sense) to the actual solution insuring that the interest solution is found in a reasonably short time.

Unfortunately, there will always exist a problem which will cause the LBL 02 guess to be far enough away from the actual solution to cause the execution time to be long. The optional routine presented below will provide an initial guess which tends to be "closer" to the actual solution than that provided by LBL 02, allowing a shorter execution time for most problems.

In use, the optional routine is executed in RAM memory and produces an initial guess for the interest. The guess is stored in register R07, and control of the calculator is transferred from the FAST routine to LBL 06 of the ROM program FI.

For the condition when PMT=0, the routine transfers to LBL 09 of the ROM program for an explicit solution. When solving for n, PV, PMT, or FV, the ROM is used in the usual manner. Don Dewey (5148) produced the mathematical expressions and wrote the program.

LBL FAST INSTRUCTIONS

1. Load the routine below into the calculator memory.

2. Go to LBL FI in the ROM.

3. Select desired status and enter known variables in the usual manner.

4. Either a) or b):

   a) solve for n, i, PV, PMT, or FV in the usual manner.

   b) Execute FAST to solve for interest using the optional routine. Do not use LBL B. The interest value is returned in the usual manner.

5. Repeat as needed from step 2.

APPLICATION PROGRAM FOR: FI

```
01*LBL "FAST"
02 9
03 STO 06
04 RCL 04
05 X=0?
06 GTO "FI"
07 6
08 STO 06
09 RCL 05
10 RCL 04
11 RCL 01
12 *
13 -
14 LASTX
15 RCL 05
16 +
17 RCL 03
18 +
19 RCL 01
20 RCL 03
21 *
22 X≠0?
23 /
24 ABS
25 STO 07
26 X<>Y
27 RCL 01
28 1
29 -
30 X↑2
31 RCL 04
32 *
33 RCL 05
34 -
35 RCL 03
36 +
37 3
38 *
39 /
40 ABS
41 RCL 05
42 X=0?
43 GTO "FI"
44 RCL 04
45 *
46 X>0?
47 GTO "FI"
48 RDN
49 STO 07
50 GTO "FI"
51 .END.
```

EQUATIONS USED IN FAST ROUTINE

If PMT*FV < 0 then FV case.
If PMT*FV >= 0 then PV case.

1. PV CASE:

$$i_0 = \left| \frac{n*PMT + PV + FV}{n*PV} \right|$$

Problem valid only if PV*PMT < 0.

2. FV CASE:

a) For PV≠0:

$$i_0 = \left| \frac{FV - n*PMT}{3*[(n-1)^2*PMT + PV - FV]} \right|$$

b) For PV=0:

$$i_0 = \left| \frac{FV + n*PMT}{3*[(n-1)^2*PMT + PV - FV]} \right|$$

FORMULAS USED IN FI

The basic financial equation used in this program was first reported in the Hewlett-Packard Journal of October 1977 (Ref. 3) where the description of its implementation in the HP-92 Financial Calculator was given. In this unique equation, all five financial variables (n, i, PV, PMT, FV) are accounted for, using the simple rule that money paid out is considered negative in sign, while money received is considered positive in sign.

The equation from page 23 of Ref. 3, is:

(1) $PV*(1+i)^n + PMT*[(1+i)^n - 1]/i + FV = 0$

Ordinary Annuity and Annuity Due Selection

In its present form, equation (1) is suitable for the ordinary annuity condition, when payments are made at the end of each period. To enable (1) to solve the annuity due condition when payments are made at the beginning of each period, a small modification is required. When this modification is added, equation (1) becomes:

(2) $PV*(1+i)^n + PMT*(1+iX)*[(1+i)^n - 1]/i + FV = 0$

where X=0 for ordinary annuity condition
X=1 for annuity due condition

When flag F09 is cleared, the ordinary annuity condition is selected. When flag F09 is set, the annuity due condition is selected. Flag F09 is toggled by LBL d.

With a simple algebraic rearrangement, (2) becomes:

(3) $[PV+PMT(1+iX)/i][(1+i)^n-1] + PV + FV = 0$

or

(4) $(PV + C)A + PV + FV = 0$

where

(5) $A = (1+i)^n - 1$

(6) $B = (1+iX)/i$

(7) $C = PMT*B$

The form of equation (4) simplifies the calculation procedure for all five variables, which are readily solved as follows:

(8) $n = LN[(C-FV)/(C+PV)]/LN(1+i)$

n is solved using LBL 01

(9) $i = [FV/PV]^{1/n} - 1$

For PMT=0, i is solved using LBL 09

For PMT≠0, i must be solved by iteration

(10) $PV = -[FV + (A*C)]/(A+1)$

PV is solved using LBL 03

(11) $PMT = -[FV + PV(A+1)]/(A*B)$

PMT is solved using LBL 04

(12) $FV = -[PV + A(PV + C)]$

FV is solved using LBL 05

Solution of Interest When PMT≠0

To solve for interest i when PMT≠0, an iterative technique must be employed, as equation (1) cannot be explicitly solved for i. This program uses Newton's Method, using exact expressions for the function of i and its derivative. The expressions are:

(13) $i_{k+1} = i_k - f(i_k)/f'(i_k)$

where

(14) $f(i) = A(PV+C) + PV + FV$

(15) $f'(i) = n*D*(PV+C) - (A*C)/i$

where

(16) $D = (1+i)^{n-1}$

(17) $= (A+1)/(1+i)$ as calculated by LBL 06

The iterative interest solving loop using equations (13), (14), and (15) starts at LBL 06.

Starting Guess For Interest

To solve for interest using Newton's Method, an initial starting guess must be provided. The program uses the following expression to provide the initial guess, i_0:

(18) $$i_0 = \left| \frac{PMT}{|PV| + |FV|} \right| + \left| \frac{|PV| + |FV|}{n^3*PMT} \right|$$

The closer the initial guess i_0 is to the actual solution i, the greater is the probability that the required solution will be obtained, and the shorter is the execution time.

Further Program Refinements

As well as being able to select either an ordinary annuity or annuity due situation, the program also enables solutions to be obtained when

a. the compounding frequency CF is not identical to the payment frequency PF, and/or,

b. interest is compounded in either discrete intervals or is continuously compounded.

When flag F08 is cleared, the discrete case is selected. When flag F08 is set, the continuous case is selected. F08 is toggled by LBL c.

Solving For n, PV, PMT, or FV.

When a solution for n, PV, PMT, or FV is required, the nominal annual interest rate i, supplied by the user, must first be converted to the effective interest rate per payment period by LBL 07. This rate, i_e, is then used by LBL 01, 03, 04, or 05 respectively to calculate the selected variable. To convert i to i_e the following expressions are used:

(19) $i_e = (1+i/CF)^{CF/PF} - 1$ (discrete case)

(20) $i_e = e^{(i/PF)} - 1$ (continuous case)

where:

i = nominal annual interest rate
i_e = effective interest rate per pmt. period
CF = compounding frequency per year
PF = payment frequency per year

Solving for Interest

When a solution for interest is required, LBL 06 (for PMT≠0) or LBL 09 (for PMT=0) produces i_e as the calculated interest value. This value of i_e must then be converted to i using LBL 11. It is the value of i, not i_e which is returned as a percentage to the X-register and register R02.

To convert i_e to i, the following expressions are used:

(21) $i = CF[(1+i_e)^{PF/CF} - 1]$ (discrete case)

(22) $i = LN[(1+i_e)^{PF}]$ (continuous case)

The common label, LBL 08

Common to all calculations is LBL 08 which is used to calculate the values of A, A+1, B, and C for use in solving the selected variable. After executing the RTN instruction following LBL 08 the stack and LAST X registers contain the following data values:

Register:	Contents:
LAST X	B
T	A+1
Z	A+1
Y	A
X	C

These values are all calculated using i_e and are then used in equations (8) to (15) as selected.

Routine Listing For: FI

```
01♦LBL "FI"
02 GTO IND 06
03♦LBL e
04♦LBL 00
05  E
06 STO 08
07 STO 09
08 CLX
09 STO 01
10 STO 02
11 STO 03
12 STO 04
13 STO 05
14 GTO 10
15♦LBL c
16 FC?C 08
17 SF 08
18 GTO 10
19♦LBL d
20 FC?C 09
21 SF 09
22♦LBL J
23♦LBL 10
24 "D"
25 FS? 08
26 "C"
27 FC? 09
28 "├E"
29 FS? 09
30 "├B"
31 ASTO X
32 RTN
33♦LBL H
34 STO 08
35 CF 22
36 RTN
37♦LBL I
38 STO 09
39 CF 22
40 RTN
41♦LBL a
42 12
43 *
44♦LBL A
45 FS? 22
46 STO 01
47 FS?C 22
48 RTN
49♦LBL 01
50 XEQ 07
51 STO Z
52 RCL 05
53 -
54 R↑
55 RCL 03
56 +
57 /
58 LN
59 RCL 07
60 LN1+X
61 /
62 STO 01
63 RTN
64♦LBL b
65 12
66 /
67♦LBL B
68 FS? 22
69 STO 02
70 FS?C 22
71 RTN
72♦LBL 02
73 RCL 03
74 ABS
75 RCL 05
76 ABS
77 +
78 RCL 04
79 X=0?
80 GTO 09
81 /
82 ABS
83 1/X
84 LASTX
85 RCL 01
86 3
87 Y↑X
88 /
89 +
90 STO 07
91♦LBL 06
92 XEQ 08
93 STO 02
94 RCL 03
95 +
96 STO Z
97 X<>Y
98 ST* 02
99 *
100 RCL 03
101 +
102 RCL 05
103 +
104 X<> Z
105 *
106 RCL 07
107 FS? 10
108 VIEW X
109  E
110 +
111 /
112 RCL 01
113 *
114 RCL 02
115 RCL 07
116 /
117 -
118 /
119 ST- 07
120 RCL 07
121 /
122  E2
123 *
124 RND
125 X≠0?
126 GTO 06
127 GTO 11
128♦LBL 07
129  E
130 RCL 02
131 %
132 RCL 08
133 RCL 09
134 FS? 08
135 X<>Y
136 RDN
137 /
138 STO 07
139 LN1+X
140 RCL 08
141 RCL 09
142 /
143 *
144 FS? 08
145 X<> 07
146 E↑X-1
```

Listing continued on page 161.

Listing continued from page 160.

Routine Listing For: FI

```
147 STO 07
148♦LBL 08
149  E
150 RCL 07
151 FS? 09
152 ST+ Y
153 /
154  E
155 RCL 01
156 RCL 07
157 LN1+X
158 *
159 E↑X-1
160 +
161 LASTX
162 RCL 04
163 R↑
164 *
165 RTN
166♦LBL 09
167 RCL 05
168 RCL 03
169 /
170 CHS
171 LN
172 RCL 01
173 /
174 E↑X-1
175 STO 07
176♦LBL 11
177 CLD
178 RCL 07
179 LN1+X
180 RCL 09
181 *
182 RCL X
183 RCL 08
184 /
185 E↑X-1
186 RCL 08
187 *
188 FS? 08
189 X<>Y
190  E2
191 *
192 STO 02
193 RTN
194♦LBL C
195 FS? 22
196 STO 03
197 FS?C 22
198 RTN
199♦LBL 03
200 XEQ 07
201 *
202 RCL 05
203 +
204 R↑
205 /
206 CHS
207 STO 03
208 RTN
209♦LBL D
210 FS? 22
211 STO 04
212 FS?C 22
213 RTN
214♦LBL 04
215 XEQ 07
216 X<> L
217 *
218 CHS
219 RCL 03
220 R↑
221 *
222 RCL 05
223 +
224 X<>Y
225 /
226 STO 04
227 RTN
228♦LBL E
229 FS? 22
230 STO 05
231 FS?C 22
232 RTN
233♦LBL 05
234 XEQ 07
235 RCL 03
236 +
237 *
238 RCL 03
239 +
240 CHS
241 STO 05
242♦LBL 12
243 END
```

Accuracy Enhancement:

The accuracy has been improved by the use of a new instruction sequence to calculate the A term:

$$(1+i)^n - 1$$

Assuming that n is stored in R01 and i (decimal) is stored in R07, the A term can be calculated in two different sequences, as follows:

Old Sequence:		New Sequence:	
RCL 07	i	RCL 07	i
1		LN1+X	LN(1+i)
+	1+i	RCL 01	n
RCL 01	n	*	n*LN(1+i)
		E↑X-1	$(1+i)^n - 1$
y^x	$(1+i)^n$		
1			
-	$(1+i)^n - 1$		

Mathematically, the two sequences produce identical results. However, over the range of numbers typically encountered, the LN1+X and E↑X-1 instructions prevent the severe loss of significant digits which occurs in the old sequence at the +1 and -1 steps. Reference 1 provides two examples for accuracy checking, as follows.

Examples for Accuracy Checking

These examples may be used to compare the accuracy of FI with other financial programs or calculators.

A.
1. Execute LBL e to clear all data registers
2. Select DISCRETE and END status (DE)
3. Key in the following variables

n = 111.1111111
%i = 2.222222222
PV = 333.3333333 ($)
PMT = 4.444444444 ($)

4. Solve for FV
The displayed FV = -5931.822943
The true FV = -5931.822944

B.
1. Execute LBL e to clear data registers
2. Select DISCRETE and END status (DE)
3. Key in the following variables

n = 63
%i = 0.000001610
PV = 0 (no need to enter this)
PMT = -1,000,000.00 ($)

4. Solve for FV
The displayed FV = 63,000,031.43 ($)
The true FV = 63,000,031.44 ($)

5. Now set FV = 0

6. Solve for PV
The displayed PV = 62,999,967.55 ($)
The true PV = 62,999,967.54 ($)

The above examples are taken from Reference 3 and are © copyright 1977, Hewlett-Packard Company. Reproduced with permission.

Simplified Solution Sequence

1. Solving for n:

   a. Calculate i_e, using equations (19) or (20), and LBL 07.
   b. Calculate A, A+1, B, and C using equations (5), (6), and (7), and LBL 08.
   c. Calculate n, using equation (8) and LBL 01.
   d. Store n in R01 and halt.

2. Solving for i:

   A. PMT=0

   a. Calculate i_e using equation (9) and LBL 09
   b. Calculate i using equations (21) or (22) and LBL 11.
   c. Store i and halt.

B. PMT≠0

a. Calculate i_0 using equation (18) and LBL 02.
b. Calculate A, A+1, B, and C using equations (5), (6), (7), and LBL 08.
c. Calculate iterative solution, using equations (13), (14), and (15) and LBL 06.
d. Exit test (i) If error is too large, back to b above
(ii) If error acceptable, continue
e. Calculate i, using equations (21) or (22) and LBL 11.
f. Store i in R02 and halt.

3. Solving for PV

a. Calculate i_e using equations (19) or (20) and LBL 07.
b. Calculate A, A+1, B, and C using equations (5), (6), and (7) and LBL 08.
c. Calculate PV using equation (10) and LBL 03
d. Store PV in R03 and halt

4. Solving for PMT

a. Calculate i_e using equations (19) or (20) and LBL 07.
b. Calculate A, A+1, B, and C using equations (5), (6), and (7) and LBL 08.
c. Calculate PMT using equation (11) and LBL 04
d. Store PMT in R04 and halt.

5. Solving for FV

a. Calculate i_e using equations (19) or (20) and LBL 07.
b. Calculate A, A+1, B, and C using equations (5), (6), and (7) and LBL 08.
c. Calculate FV using equation (12) and LBL 05
d. Store FV in R05 and halt.

LINE BY LINE ANALYSIS OF FI

Lines 01-02 provide access to any FI subroutine that begins with a numeric label.

Lines 03-14 clear the financial registers R01-R05 and set CF=PF=1 in R08 and R09. Line 14 is a jump to the status display.

Lines 15-18 toggle flag F08 which controls Continuous/Discrete compounding. Line 018 is a jump to the status display.

Lines 19-21 toggle flag F09 which controls the Begin/End switch.

Lines 22-32 are the status display which shows one of the codes CE, CB, DE, DB.

Lines 33-36 store the number of compounding periods in R08 = CF = compounding frequency.

Lines 37-40 store the number of payment periods in R09 = PF = payment frequency.

Lines 41-43 multiply the X-register by 12 before entering the LBL A routine.

Lines 44-48 either store n in R01 and stop, or drop through to line 049.

Lines 49-63 solve for n via formula (8) and store n in R01.

Lines 64-66 divide the X-register by 12 before entering the LBL B routine.

Lines 67-71 either store %i in R02 and stop, or drop through to line 072.

Lines 72-127 are the major part of the program which solves for %i. Line 79 tests whether %i can be calculated directly if PMT=0. Otherwise lines 73-90 compute the initial guess for %i via formula (18). Lines 91-126 are the recurrence loop for formulas (13)-(17). Formula (14) is complete at line 103 and formula (15) is complete at line 117. Line 127 is a branch to complete the calculation of %i.

Lines 128-165 are a special subroutine. Line 148 provides access to a second entry point within the subroutine. Lines 129-147 calculate formula (19) or (20). Lines 148-165 calculate formulas (5), (6), and (7) which are constants used by other parts of the program.

Lines 166-175 calculate formula (9) for %i when PMT=0.

Lines 176-193 finish the calculation of %i and restore the rate by calculating formula (21) or (22).

Lines 194-198 either store PV in R03 and stop, or drop through to line 199.

Lines 199-208 calculate PV via formula (10) and store PV in R03.

Lines 209-213 either store PMT in R04 and stop, or drop through to line 214.

Lines 214-227 calculate PMT via formula (11) and store PMT in R04.

Lines 228-232 either store FV in R05 and stop, or drop through to line 233.

Lines 233-241 calculate FV via formula (12) and store FV in R05.

Line 242 is provided to allow a running program to stop so the program pointer is in ROM.

NUMERIC LABELS/FUNCTIONS IN THE FI PROGRAM

Although FI is a complete self-contained program, some users may wish to use some of FI's subroutines in their own programs. The following list gives a correspondence between numeric labels and subroutines to be called as part of FI programs. To call a subroutine from one of your own programs, first store the number corresponding to the desired function in data register R06. Then use the instruction XEQ "FI" as part of your program. The execution times in seconds for the various subroutines are in parentheses in the following list.

Numeric Label Number in R06	Keyboard Label	Subroutine Function
00	e	Clear R01-R05 and store 1 in R08 and R09 (<1 sec.)
01	A	Solve for n (3.5 sec.)
02	B	Solve for %i (variable)
03	C	Solve for PV (2.5 sec.)
04	D	Solve for PMT (3.3 sec.)
05	E	Solve for FV (3.2 sec.)
12	None	Serves only to restore keyboard functions to top rows of keys

The following special comments apply to FI subroutines that would be called from other programs.

First note that labels 01-05 only solve for the indicated variables, whereas the keys A-E perform the double functions of either solving or storing values. If you need to store the value of a financial variable in one of R01-R05 then your program should do that directly. Subroutines for storing are neither necessary nor provided.

The purpose of label 12 at the end of the program is to allow a running program to stop so the program pointer is in ROM and the automatic local label key assignments of the FI functions on the top row of keys will be restored to those keys. Otherwise, a running program would normally stop and leave the program pointer in RAM which would make the top row key assignments "disappear".

Numeric labels 07,08,09 & 11 are intended to be internal subroutines within FI which would seem to perform no useful purpose outside of FI. However, the truly curious PPC member may be able to jump into the middle of FI by calling these routines as "hidden functions". See the line by line analysis for the purpose of these functions. Example 11 in the FI documentation uses label 07 in this manner.

REFERENCES FOR FI

1. W.L. Crowley and F. Rode, "A Pocket-Sized Answer Machine For Business and Finance," Hewlett-Packard Journal, May 1973.

2. R.B.Neff and L. Tillman, "Three New Pocket Calculators: Smaller, Less Costly, More Powerful," Hewlett-Packard Journal, November 1975.

3. Roy E. Martin, "Printing Financial Calculator Sets New Standards For Accuracy and Capability," Hewlett-Packard Journal, October 1977.

4. Greynolds, Aronofsky, Frame, "Financial Analysis Using Calculators," McGraw-Hill, 1980.

For anyone wishing to further his/her knowledge on the subject of financial analysis, and as it applies to calculators, Reference 4 is probably the most definitive book available to date. The 470-page volume assumes the reader has no previous knowledge of financial analysis, and commences at an elementary level, taking the reader through the theory and practice. The main subjects covered are:

a. Basic Concepts in Compound Interest
b. Simple Annuities
c. General Annuities
d. Continuous Compounding and/or Payments
e. Variable Cash Flows and Internal Rate of Return
f. Balloon Annuities Using Present Values
g. Special Applications

CONTRIBUTORS HISTORY FOR FI

Graeme Dennes (1757) is responsible for programming the accuracy enhancements in FI and for writing the first substantial version of FI, after analyzing the problems associated with the various formulas involved.

The addition of the general annuity capability and an improvement to the initial guess routine were produced by Don Dewey (5148). Don also tested and further debugged the program using literally thousands of test examples. There were more changes made until the program evolved to its final form. Cliff Carrie (834) suggested changes in the use and placement of the local numeric labels. Graeme Dennes and Don Dewey are to be credited with the lion's share of the work on FI, both in programming and in the documentation.

FINAL REMARKS FOR FI

The first dedicated hand-held financial calculator was the HP-80 (Reference 1). This was later followed by the HP-22 (Reference 2) and HP-27, although they never replaced the HP-80.

As users accepted the new calculators, it wasn't long before they began to demand more in facilities, capabilities and accuracy. These needs were readily satisfied by the HP-92 printing financial calculator (Reference 3).

The HP-92 article provided the inspiration and the starting point for the FI program by setting challenges for accuracy and execution times. It also made available for the first time a single unique financial equation which accounts for all five basic financial variables.

Commencing with this single equation, the FI program was conceived as a new, original approach to providing a highly accurate and fast financial program for the HP-41C calculator. Both the facilities and mathematical approaches of FI have not been used together in any previous HP calculator program. Financial programs are in other HP-41C ROM's but the capability and accuracy justify including the FI program as part of the PPC ROM.

By applying a reformulation and simplification to the mathematics of the new HP financial equation, the execution times and accuracy have been improved over all previous programs. The execution times when solving for interest rate have been reduced by the use of a simplified, although accurate, Newton's Method for fast convergence, coupled with a routine which produces an initial starting guess "close" to the exact solution.

Previous financial programs used standard instruction sequences to calculate terms of the form

$$(1+i)^n - 1$$

as described elsewhere. Often, several different terms of this type were used in one program. The calculation of this term created most of the error in those programs. The high accuracy of FI was achieved by calculating this term using the LN1+X and E↑X-1 instructions, and calculating it only once per solution.

Thanks to the Hewlett-Packard Company for allowing the reproduction of numerical examples from Reference 3.

FURTHER ASSISTANCE ON FI

Don Dewey (5148) phone (415) 526-1677 evenings
Graeme Dennes (1757) phone (415) 592-2957 evenings
Gary M. Tenzer (1816) phone: (213) 557-8336

NOTES

TECHNICAL DETAILS

XROM: 10, 63 | **FI** | SIZE: 010

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: 2 char. status
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
- R00: not used
- R01: n
- R02: %i
- R03: PV
- R04: PMT
- R05: FV
- R06: function call #
- R07: %i as decimal
- R08: CF
- R09: PF

N.B. R02 used as temporary store when solving for interest.

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: set=continuous clear=discrete
- 09: set= BEGIN clear=END
- 10: set=view iterations clear=no display
- 25: not used

Display Mode:
FIX 2 recommended

Angular Mode:
not used

Unused Subroutine Levels:
4

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine:
A, B, C, D, E, H, I, J
a, b, c, d, e
00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12

Execution Time: See NUMERIC LABELS section in the FI documentation

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: FI

Bytes In RAM: 324

Registers To Copy: 47

Other Comments:

CONTINUED FROM PAGE 143

MEM	QUAN	ORD
4516	1	298
4518	1	129
4520	3	766
		767
		768
4521	1	817
4522	1	1850
4523	1	1953
4525	2	2301
		2495
4528	1	49
4530	1	2091
4532	1	133
4534	1	1444
4537	1	206
4538	1	1602
4542	1	1437
4544	1	1220
4545	1	29
4550	1	424
4551	2	1431
		2486
4552	1	2411
4554	1	1137
4555	1	1621
4559	3	59
		60
		2009
4561	1	1945
4564	1	1505
4565	2	50
		51
4567	1	1585
4573	3	414
		415
		416
4575	1	895
4582	1	246
4584	3	1401
		1402
		1403
4588	1	1233
4591	1	234
4593	1	1849
4596	1	21
4597	1	1714
4598	3	787
		788
		789
4600	2	2135
		2136
4601	1	2296
4602	1	732
4603	2	1242
		1243
4605	1	1099
4608	1	128
4609	1	2225
4610	1	281
4613	1	795
4615	1	987
4617	1	431
4621	1	347
4623	1	519
4624	2	911
		2264
4626	1	337
4627	1	821
4628	1	68
4629	1	645
4631	1	824
4632	1	1926
4635	1	534

MEM	QUAN	ORD
4636	1	883
4638	1	34
4639	1	1785
4641	2	190
		1241
4642	1	176
4643	1	664
4644	1	1157
4646	1	393
4647	1	2043
4648	1	1617
4650	1	1370
4652	1	1179
4654	1	459
4657	1	23
4659	1	2418
4661	1	841
4667	1	2164
4669	2	609
		610
4670	1	341
4671	1	628
4673	1	62
4676	4	1151
		1152
		1153
		1901
4677	1	1154
4681	1	474
4684	1	2401
4685	1	2158
4689	1	1651
4690	1	1568
4691	1	1549
4692	1	1904
4694	1	1557
4695	1	16
4696	1	964
4697	1	1201
4704	1	4
4705	1	192
4706	1	2256
4710	2	1033
		1034
4712	1	856
4714	1	1610
4715	1	423
4716	1	1848
4717	1	2145
4719	1	1564
4725	4	1479
		1480
		1481
		1482
4727	1	222
4728	2	1057
		1058
4729	1	2308
4732	1	226
4733	1	906
4734	1	282
4735	1	998
4736	2	753
		1742
4737	1	1554
4739	1	604
4740	1	813
4741	1	1203
4745	1	1704
4748	1	1628
4751	1	156
4755	1	204
4756	3	1521

MEM	QUAN	ORD
		1522
		1523
4760	1	1784
4764	1	105
4766	1	1087
4777	1	584
4780	1	1055
4783	1	889
4785	1	580
4787	1	1317
4789	1	1720
4790	1	1711
4791	1	450
4794	1	943
4796	3	588
		589
		590
4800	1	1130
4801	1	2395
4806	1	2069
4807	3	684
		685
		686
4809	1	488
4812	2	2112
		2113
4816	1	799
4818	2	1024
		1025
4823	1	983
4825	1	1448
4826	1	1413
4829	1	1001
4831	2	2183
		2184
4832	1	1540
4833	1	575
4834	1	1144
4836	1	860
4841	1	1059
4843	1	2200
4845	3	1222
		1223
		1224
4848	1	629
4849	1	995
4852	1	486
4854	1	873
4856	1	311
4859	1	890
4861	1	639
4866	1	1647
4869	1	644
4870	1	1311
4871	1	770
4872	1	1212
4873	1	937
4876	1	2302
4877	1	1710
4878	1	1607
4880	1	447
4881	2	138
		139
4884	1	1457
4889	1	748
4891	2	403
		2219
4893	1	1244
4898	1	433
4901	1	2431
4909	1	2252
4912	1	957

MEM	QUAN	ORD
4913	1	1829
4914	1	1910
4916	1	1840
4918	1	2026
4919	1	1003
4920	1	2023
4921	1	1730
4922	1	1981
4923	1	1230
4924	1	1596
4925	1	481
4926	1	1145
4928	1	1648
4929	1	970
4930	1	825
4932	1	368
4938	1	877
4939	1	1184
4940	1	1313
4943	1	1441
4944	1	656
4946	1	1251
4947	1	808
4949	1	1110
4950	1	1721
4953	1	1833
4954	2	2011
		2012
4957	1	1434
4965	1	303
4968	1	704
4970	1	1917
4973	1	84
4975	1	845
4976	1	1231
4978	1	1097
4979	1	275
4982	2	862
		863
4983	1	124
4985	2	853
		854
4987	2	1576
		1577
4988	1	1893
4991	1	835
4995	1	1646
4997	1	1552
5000	1	1177
5003	1	440
5004	1	389
5007	1	1358
5011	2	1518
		1519
5013	3	572
		573
		574
5016	1	2249
5017	1	2167
5024	1	1466
5025	1	324
5026	1	1323
5031	1	14
5034	1	791
5037	1	2132
5043	1	1226
5045	1	776
5047	3	771
		2195
		2196
5052	1	683
5053	2	1856
		1857

MEM	QUAN	ORD
5055	1	1207
5056	1	2311
5057	1	1061
5060	1	367
5061	1	1045
5064	1	406
5065	1	247
5066	1	1808
5071	1	1128
5073	2	980
		2485
5075	1	1412
5078	3	242
		1227
		1228
5079	1	521
5080	1	413
5084	1	1816
5086	1	132
5088	1	1089
5091	3	408
		409
		1298
5092	1	540
5094	1	1527
5097	1	217
5101	1	634
5104	2	264
		265
5107	1	2463
5108	1	1982
5110	1	568
5115	1	2226
5116	1	480
5117	1	2220
5121	1	2259
5122	2	239
		1062
5125	1	1757
5128	1	1167
5136	1	954
5137	1	400
5138	1	366
5140	1	1188
5144	3	294
		295
		296
5145	1	1506
5147	2	677
		678
5148	1	810
5150	1	203
5151	2	1514
		1515
5152	2	1511
		1512
5153	1	1724
5154	1	2172
5156	1	463
5161	1	723
5165	1	1769
5168	1	2174
5169	2	161
		2165
5170	1	1755
5171	2	67
		1335
5173	1	22
5174	1	1817
5176	1	847
5178	1	73
5182	1	2379
5183	3	879

MEM	QUAN	ORD
		880
		881
5184	1	1017
5187	1	516
5189	2	1536
		1537
5193	1	262
5194	1	1032
5196	1	1713
5199	1	1858
5204	1	1324
5205	1	2394
5209	1	975
5211	2	343
		344
5214	1	1952
5215	1	2089
5217	2	121
		122
5221	1	1397
5223	1	412
5224	3	891
		892
		893
5227	1	283
5228	1	372
5231	2	1760
		2216
5236	1	772
5240	1	1619
5242	1	1851
5245	1	1229
5246	2	1284
		1285
5248	1	736
5252	3	545
		546
		547
5254	1	2079
5255	1	1656
5256	1	273
5258	1	1507
5266	1	717
5267	1	1102
5268	1	17
5269	1	1776
5270	1	1411
5271	1	1556
5275	2	2087
		2088
5277	1	1135
5281	1	177
5285	1	438
5286	1	497
5287	1	371
5289	3	1474
		1475
		1476
5290	1	42
5291	1	198
5292	1	120
5294	1	2120
5295	1	1277
5298	2	342
		1471
5302	1	2384
5304	2	1365
		1366
5307	1	790
5308	1	1642
5310	1	111
5313	1	1820

MEM	QUAN	ORD
5314	3	1196
		1197
		1198
5317	1	822
5318	1	1547
5320	1	76
5328	1	1869
5329	1	25
5330	1	1090
5333	1	1644
5336	2	720
		721
5339	1	769
5345	1	1329
5346	1	263
5348	1	175
5349	1	1312
5350	1	809
5353	2	1214
		1215
5355	1	2344
5356	1	909
5358	1	258
5359	1	272
5360	1	457
5361	1	319
5363	2	1495
		1496
5366	2	696
		697
5371	1	1567
5376	1	1029
5379	11	1637
		1638
		1639
		1640
		1641
		2275
		2276
		2277
		2278
		2279
		2280
5384	1	2059
5385	1	1894
5386	2	1854
		1855
5389	1	1338
5390	2	47
		48
5392	1	785
5393	1	1091
5395	1	1454
5396	1	1395
5397	1	1443
5398	1	35
5401	1	616
5404	1	852
5406	1	1103
5407	1	1237
5412	2	352
		353
5413	1	2154
5415	1	2020
5416	2	1080
		1081
5419	2	773
		774
5422	1	1261
5423	1	495
5425	1	1189
5426	1	1815
5428	1	228

APPENDIX F CONTINUED ON PAGE 169

FL - FLAG INPUTS FOR LB

FL like its 'brothers' BL and XL are 'program aid' type programs intended to be executed from the keyboard. Two of these programs cannot be called as a subroutine because successive inputs are required that can't be 'pre-loaded' or outputs are separated by a STOP instruction.

Example 1: The following flags are to be set during initialization of a program.

Set Flag	Remarks
5	Program loop control
25	System error flag
26	Audio Enabled
28	Decimal Radix
39	"0" Digits
40	FIX MODE
44	Continuous ON

The following example shows a fix 1 display. Key flag inputs in numerical order.

DO:	SEE:	RESULT:
XEQ FL	0.0	Start of program
5, R/S	-5.0	Negative output, key next flag
25, R/S	4.0	Tone sounded, first LB byte
R/S	0.0	Tone, second LB byte
R/S	0.0	Tone, third LB byte
R/S	-25.0	Negative output, key next flag
26, R/S	-26.0	Negative output, key next flag
28, R/S	-28.0	Negative output, key next flag
39, R/S	104.0	Tone, Fourth LB byte
R/S	1.0	Tone Fifth LB byte
R/S	-39.0	Negative, key next flag
40, R/S	-40.0	Negative, key next flag
44, R/S	-44.0	Negative, all desired flags input
56, R/S	136.0	Tone, sixth LB byte
R/S	0.0	Tone, seventh LB byte
R/S	-56.0	Desired result obtained, program finished.

FL

BACKGROUND FOR FL

One of the most useful programs in the PPC ROM is LB. The ability to quickly, conveniently, and confidently enter arbitrary byte sequences into 41 memory makes the ROM worthwhile for this program alone. (Thanks again Roger). The user, however, needs guidance in determining what bytes should be loaded to produce the desired instructions in memory. The plastic HEX/DECIMAL Byte Table included with this manual provides the byte number for simple instructions. The more complex instructions such as LBL'S, END'S and Text lines are described in the LB program documentation. Three 'special' types of byte sequences however, are 'coded' in a manner complex enough to invite error and take a significant amount of time. These are outlined in the table below.

Description	Use	Remarks
Text line to set all desired flags	FL	RCL M, STO d
Text line for BLDSPEC	BL	RCL M, ACSPEC, PRBUF
XROM numbers	XL	two XROM bytes (in memory)

These three programs were added to the ROM as an aid to the user in preparing instructions to be entered into memory. The input format, speed, and code efficiency is not neccessarily optimized. They were "quick and dirty" last minute routines added to round out the support for LB --which took a significant percentage of the total bytes. Because of the time limitations to write these programs they have limits and weakness. PPC members will probably explore them and, hopefully extend their use. FL, for example doesn't 'know' when it is finished. You simply start a new 8 Flag group and complete the seventh byte by keying in a non-existent flag, flag 56.

COMPLETE INSTRUCTIONS FOR FL

FL is used by first executing FL, and keying in the flag numbers using R/S. The routine is not automatic and depends on the user to continue by pressing R/S, or keying another flag and then pressing R/S.

Rule: -NN: If display shows the previously input flag number as a negative number key next flag, R/S.

The display may occasionally show a number other than a negative of the previously entered flag number or LB byte. This will usually happen when the first flag of a new 8 flag group is entered.

MORE EXAMPLES OF FL

Example 2: A new 41 user decides to use synthetic programming to set Flags 6-10. Before using FL, however, he examines the byte count.

Synthetic	Standard
"Text line"	SF 06
RCL M	SF 07
STO d	SF 08
	SF 09
	SF 10
12 Bytes	10 Bytes

He concludes that a synthetic text line stored in the flag register requires a constant 12 bytes and that this approach is suitable only under the following conditions.

A. Six or more flags are to be set.

B. Flags not controllable are to be set. (Remember that every call to IF is 3 or 4 bytes-two for call, 1 or 2 for flag number.)

Example 3: A program requires the following for proper execution.

Flag 1 set for program use	Flag 1 set
Don't print	Flag 21 clear
Bender (beeper) turned off	Flag 26 clear
User Mode	Flag 27 set
Comma Radix	Flag 28 clear
FIX 3	Flags 38 & 39 & 40 set.
RAD Mode	Flag 43 set
Continuous ON	Flag 44 set
Printer "Nonexistent" for speed	Flag 55 clear

FLAGS TO BE SET: 1, 27, 38, 39, 40, 43, 44.

DO:	SEE:	RESULT:
XEQ FL	0.0	Start of program
1, R/S	-1.0	Negative, key next flag
27, R/S	64.0	TONE, First LB Byte
R/S	0.0	TONE, Second LB Byte
R/S	0.0	TONE, Third LB Byte
R/S	-27.0	Negative, key next flag
38, R/S	16.0	TONE, Fourth LB Byte
R/S,	-38.0	Negative, key next flag
39, R/S	3.0	TONE, Fifth LB Byte
R/S	-39.0	Negative, key next flag
40, R/S	-40.0	Negative, key next flag
43, R/S	-43.0	Negative, key next flag
44, R/S	-44.0	Negative, key next flag, NO MORE! Use 56.
56, R/S	152.0	TONE, Sixth LB Byte
R/S	0.0	TONE, Seventh LB Byte
R/S	-56.0	Concluding "check".

The text line of 7 bytes requires a text 7 byte ahead of the flag bytes. The LB inputs for the text line would be:

247, 64, 0, 0, 16, 3, 152, 0

This line displays as ᵀ @ ⁻ ⁻ ⊠ ⊠ ⊠ ⁻ and prints as @✦✦8←ö✦ . The program segment to set the seven flags would be:

"@ ⁻ ⁻ ⊠ ⊠ ⊠ ⁻	247, 64, 0, 0, 16, 3, 152, 0	(8 bytes)
RCL M	144, 117	(2 bytes)
STO C	145, 126	(2 bytes)
		12 bytes

FURTHER DISCUSSION OF FL

The d register is used by the 41C 'operating system' to store the status of the system flags. The d register, like all 41 registers is 56 bits. Each flag is represented by a bit, which if represented by a 1 for set, 0 for clear would be a string of 56 1's and zeros. Using the flags of Example 3, the 56 bits would be

Flag 0 → ... ← Flag 31

BIN	0100 0000	0000 0000	0000 0000	0001 0000
HEX	4 0	0 0	0 0	1 0
DEC	64	0	0	16

FLAG 32 → ... ← Flag 55

BIN	0000 0011	1001 1000	0000 0000
HEX	0 3	9 8	0 0
DEC	3	152	0

LINE BY LINE ANALYSIS OF FL

Lines 21 through 25 clears the alpha register and the stack. The flag number (input) is entered by the user at line 25. Line 25 replicates the input and the 8 of line 27 and XROM QR of line 28 converts the flag input (0 to 55) into a byte number (0 to 6) and a flag number (0 to 7) within that byte. The flag number (MOD 8) is placed in the M register and the previous byte number taken out of M by the X exchange M of

T E C H N I C A L D E T A I L S

XROM: 10,43 | **FL** | SIZE: 000

Stack Usage:
- 0 T: USED
- 1 Z: USED
- 2 Y: USED
- 3 X: USED
- 4 L: USED

Alpha Register Usage:
- 5 M: USED
- 6 N: USED
- 7 O: USED
- 8 P: USED

Flag Usage: NONE
04:
05:
06:
07:
08:
09:
10:
25:

Other Status Registers:
- 9 Q:
- 10 ⊦:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

NONE USED BY ROUTINE

Display Mode: N/A

Angular Mode: N/A

Unused Subroutine Levels: 4*

ΣREG: NOT USED

Data Registers:
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

NONE USED BY ROUTINE

Global Labels Called:

Direct	Secondary
QR	NONE

Local Labels In This Routine:
LBL 00
LBL 01
LBL 13
LBL 14

Execution Time: 0.2 - 1.2 seconds per input.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	YES
Program File:	BL
Bytes In RAM:	62
Registers To Copy:	46

Other Comments:
* This routine is not intended to be called as a subroutine.

line 30. If the current flag input is not in the previous byte, execution is transferred to LBL 13 (line 47) by the logic compare at line 31. The LBL 13 routine displays the previous byte. If the branch is not made to label 13, the flag number (MOD 8) is brought back to X at line 33 and 2^{7-f} is calculated by lines 34-39 and added to the running count for the current byte in the N register by line 40. If $2^{7-f} = 1$ (the E in line 40 is "1") the flag input is the last flag in the byte and LBL 14 is executed to display the result. After processing by LBL 14 the flag number is brought back to X at line 44 and made negative before execution is halted for the next input by the GTO in line 46 and the STOP at line 25.

LBL 14 (line 53) displays the current byte number by pulling it out of the N register (clearing it at the same time) at lines 54 and 55 and incrementing the current byte number in M at line 56. Line 57 is a text zero "NOP" that makes the ISG M, NOP, a three byte 1, STO +M operation that doesn't require use of the stack. At this point a tone 7 is produced at line 58 and the routine stops to display the current byte number in X.

LBL 13 (line 47) saves the current flag number and base 8 decomposition in T, Y, and Z and calls LBL 14 to display the previous byte. The GTO 01 at line 52 sets up a test to determine whether another byte is ready to be displayed.

Routine Listing For: **FL**

```
21♦LBL "FL"      41 E
22 CLA           42 X=Y?
23 CLST          43 XEQ 14
24♦LBL 00        44 R↑
25 STOP          45 CHS
26 RCL X         46 GTO 00
27 8             47♦LBL 13
28 XROM "QR"     48 X<> [
29♦LBL 01        49 ENTER↑
30 X<> [         50 XEQ 14
31 X≠Y?          51 RDN
32 GTO 13        52 GTO 01
33 X<> [         53♦LBL 14
34 7             54 CLX
35 X<>Y          55 X<> \
36 -             56 ISG [
37 2             57 ""
38 X<>Y          58 TONE 7
39 Y↑X           59 STOP
40 ST+ \         60 RTN
```

CONTRIBUTORS HISTORY FOR FL

Keith Jarett (4360) wrote FL for the PPC ROM to support LB.

FURTHER ASSISTANCE ON FL

Call Keith Jarett (4360) at (213) 374-2583 EVE.
Call Richard Nelson (1) at (714) 754-6226.

NOTES

CONTINUED FROM PAGE 163

MEM	QUAN	ORD
5430	1	633
5432	1	1516
5435	1	1290
5436	1	603
5437	1	184
5439	1	1148
5443	1	675
5447	1	219
5452	3	2141
		2142
		2143
5453	1	1709
5454	1	512
5460	1	1948
5461	1	807
5462	1	63
5464	1	1853
5465	2	364
		365
5467	1	897
5468	1	1754
5469	3	727
		728
		729
5470	1	601
5472	1	453
5477	1	831
5478	1	905
5480	1	1119
5482	3	826
		827
		828
5485	1	803
5488	1	55
5489	1	941
5491	1	2483
5492	1	2436
5495	1	200
5497	1	734
5498	1	2214
5499	1	1676
5500	1	1606
5502	1	2029
5503	3	1266
		1267
		1268
5506	1	349
5508	1	115
5509	2	1275
		1276
5510	1	1565
5511	2	605
		1978
5514	3	709
		710
		711
5517	1	1500
5519	1	920
5520	1	421
5522	1	786
5523	1	505
5524	3	2161
		2162
		2163
5525	1	515
5526	1	54
5528	1	276
5530	1	2000
5531	1	1210
5533	1	1147
5534	1	594
5535	3	383
		384

MEM	QUAN	ORD
		692
5536	1	1763
5537	1	2045
5538	1	690
5541	1	760
5542	1	79
5544	1	942
5545	1	718
5546	1	12
5547	1	359
5548	3	316
		317
		318
5551	1	1681
5552	1	465
5553	1	235
5555	2	92
		93
5556	1	208
5557	2	193
		194
5558	1	1722
5560	1	1590
5564	1	606
5569	1	1455
5572	3	38
		39
		40
5573	1	1545
5574	2	81
		82
5575	1	908
5576	1	2152
5577	1	210
5578	1	1204
5579	2	286
		1304
5581	1	386
5585	1	2119
5586	1	332
5587	1	1764
5591	1	88
5592	2	702
		703
5593	1	1618
5594	1	1213
5596	1	1028
5597	1	796
5598	1	705
5602	1	1139
5603	2	2207
		2208
5607	1	687
5608	8	868
		913
		914
		915
		916
		917
		918
		919
5609	1	1175
5612	1	647
5613	1	1429
5615	1	150
5618	1	740
5621	2	231
		232
5623	1	149
5625	1	885
5626	1	1661
5628	1	792
5629	2	1076

MEM	QUAN	ORD
		1077
5634	1	144
5636	1	382
5637	3	313
		314
		315
5638	4	2115
		2116
		2117
		2118
5639	1	361
5641	1	1149
5642	1	1002
5644	1	1319
5646	1	614
5648	1	2129
5650	1	346
5651	1	185
5654	3	1018
		1019
		1020
5655	1	2062
5656	1	458
5657	1	669
5658	1	1790
5659	1	600
5661	1	665
5662	1	1472
5664	1	1600
5666	1	237
5667	1	356
5668	1	2316
5669	1	1615
5670	1	1247
5671	1	227
5673	1	1693
5674	1	1624
5675	1	1008
5680	2	428
		429
5686	1	851
5687	1	1379
5688	1	2251
5689	1	712
5691	1	1134
5694	1	2450
5695	1	850
5696	1	250
5697	1	1627
5698	1	1225
5699	1	1841
5700	1	1166
5701	1	1874
5702	1	325
5703	1	550
5704	1	921
5706	1	289
5708	1	1548
5709	1	432
5710	1	470
5711	1	274
5713	1	1318
5716	1	1461
5719	3	159
		1821
		1822
5722	1	350
5723	1	643
5724	1	1010
5725	1	496
5726	1	360
5728	1	399
5729	1	2263

MEM	QUAN	ORD
5730	1	1566
5734	1	756
5737	1	1256
5738	1	1531
5739	1	1581
5740	2	339
		2133
5742	1	635
5743	1	1316
5747	1	2204
5750	1	80
5751	1	564
5752	1	1560
5753	1	1509
5754	3	832
		833
		834
5755	1	1595
5756	1	586
5757	1	101
5758	1	1436
5759	1	823
5760	1	1620
5761	1	988
5762	1	165
5763	1	2108
5764	1	230
5765	1	18
5767	2	215
		1337
5771	1	2433
5772	1	701
5773	1	489
5778	1	1768
5781	7	563
		1731
		1732
		1733
		1734
		1735
		1736
5782	1	1164
5783	2	1191
		1192
5786	1	370
5787	1	2203
5789	1	866
5790	1	2182
5793	1	557
5795	1	529
5799	1	539
5800	1	939
5801	1	934
5802	1	596
5803	1	1303
5804	1	110
5805	1	1750
5807	1	695
5813	1	475
5815	2	2109
		2110
5816	2	1255
		1489
5817	1	1473
5818	1	99
5819	1	1438
5822	1	410
5823	2	864
		882
5824	2	971
		1270
5827	2	1367
		1368

MEM	QUAN	ORD
5828	1	1513
5830	1	1723
5833	1	994
5834	1	1404
5836	1	446
5838	1	1306
5839	1	1717
5841	2	955
		956
5843	1	804
5845	1	312
5846	1	1923
5849	1	461
5850	1	661
5851	1	1592
5853	1	1452
5854	1	798
5856	1	537
5857	1	1359
5859	1	2231
5863	1	1780
5864	1	743
5865	1	1353
5866	2	967
		968
5867	1	1762
5868	1	691
5871	1	1347
5872	2	973
		974
5874	1	1442
5875	1	396
5876	1	420
5877	1	1042
5878	1	1460
5879	1	1269
5880	1	1075
5881	1	1599
5882	1	357
5883	2	699
		700
5884	1	397
5885	1	1218
5887	3	471
		472
		473
5888	1	688
5889	2	464
		775
5890	1	681
5891	1	2343
5892	1	1348
5894	4	1751
		1771
		1772
		1773
5895	1	640
5897	1	747
5898	1	1873
5905	1	509
5907	1	693
5908	1	354
5911	1	2388
5912	1	1282
5914	1	454
5920	1	1328
5923	1	1344
5924	1	961
5925	1	460
5926	1	1453
5927	1	1988
5928	1	839
5931	1	551

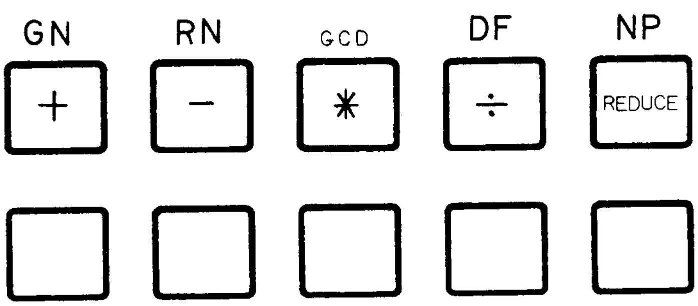
MEM	QUAN	ORD
5932	1	499
5933	1	587
5934	1	1490
5937	1	504
5939	1	508
5941	1	279
5943	1	2070
5944	4	1082
		1083
		1084
		1085
5945	1	1257
5946	1	818
5947	1	1325
5951	1	1217
5952	1	2114
5953	1	541
5954	1	1979
5958	2	1006
		1007
5959	1	730
5962	1	2255
5964	1	493
5965	1	812
5967	1	1696
5968	1	1477
5969	1	1113
5970	1	1143
5978	1	2053
5979	1	1486
5980	1	626
5982	1	962
5983	1	630
5984	1	722
5985	1	1598
5987	1	894
5990	1	1023
5991	2	875
		876
5993	2	1132
		1133
5995	1	1643
5996	1	2157
5998	1	1445
5999	1	1485
6000	1	896
6001	1	925
6003	1	814
6004	1	2213
6005	2	1487
		1488
6006	1	901
6007	1	1292
6009	1	978
6010	1	819
6013	1	1122
6014	1	1908
6016	2	2265
		2266
6018	1	1380
6020	1	855
6021	1	1470
6024	1	1844
6026	1	907
6027	1	1209
6030	1	953
6032	1	1095
6033	1	1078
6034	1	820
6035	1	1054
6036	1	837
6037	1	1346
6045	1	1399

MEM	QUAN	ORD
6046	1	923
6047	1	1126
6048	1	1005
6049	1	598
6050	1	1360
6051	1	902
6052	1	1497
6053	1	1016
6056	1	1219
6058	1	1240
6059	1	1096
6061	2	1308
		1867
6062	1	1384
6063	1	1392
6064	1	1994
6067	1	1725
6068	1	1281
6069	1	1302
6070	1	1458
6071	1	542
6073	1	1712
6075	1	562
6076	1	1100
6077	1	1842
6078	1	912
6079	1	2461
6082	1	1439
6083	1	1208
6085	1	707
6086	1	708
6089	1	874
6090	1	713
6094	1	938
6097	1	2367
6098	2	992
		993
6100	1	952
6101	1	1073
6106	1	1027
6107	1	1038
6110	1	1741
6111	2	1446
		1447
6113	1	2127
6115	1	1056
6117	1	1109
6119	1	1716
6121	1	990
6124	1	910
6125	1	2015
6127	1	1262
6128	1	1286
6129	1	1860
6130	2	1314
		1315
6132	1	1582
6133	2	1974
		1975
6134	1	1464
6135	3	1541
		1649
		1650
6137	1	1465
6138	1	1580
6140	2	1574
		1575
6141	3	1583
		1584
		1683
6143	1	1405
6144	1	1691
6145	1	1111

APPENDIX F CONTINUED ON PAGE 173

FR - FRACTIONS

This program has routines for addition, subtraction, multiplication, division, and reduction of fractions. Setting a flag allows fractional forms to be displayed in the alpha register. The functions available on the top row of keys on the keyboard are indicated in the following diagram.



These same functions are referenced in the examples and instructions by enclosing the function name in square brackets [ ].

Example 1: Use FR to reduce the fraction 1547/3757.

Key GTO "FR" and go into USER mode. This step puts the program counter in ROM and makes the fraction functions available on the top row of keys. Set flag 10 to display fractions in the alpha register. (You may also wish to clear flag 29). To reduce a fraction key numerator ENTER↑ denominator and push [REDUCE]. For this example key 1547 ENTER↑ 3757 and push [REDUCE]. See 7/17 in the display. The numerator 7 and the denominator 17 are left in the Y and X registers respectively.

Example 2: Use FR to subtract 5/18 - 19/24.

Key 5 ENTER↑ 18. This puts the first fraction in the machine. Continuing, key ENTER↑ 19 ENTER↑ 24. Both fractions should now be on the stack as:

```
T:  5
Z: 18    first fraction
Y: 19
X: 24    second fraction
```

Pressing [-] displays the answer -37/72 and leaves the answer on the stack as:

```
T:  *
Z:  *
Y: -37
X:  72
```

COMPLETE INSTRUCTIONS FOR FR

(Keyboard Operations)

1) Key GTO "FR". The keyboard functions will now be available on the top row of keys.

2) Flag 10 controls a display option. If F10 is set then answers will be displayed in a fractional form using the alpha register. (It may also be desirable to clear flag 29) If F10 is clear the alpha register is not used.

3) Regardless of the state of flag 10 all fractional answers are left in reduced form with the numerator in Y and the denominator in X. Chaining of further operations may continue by using previous answers.

4) To reduce a fraction key numerator ENTER↑ denominator and push [REDUCE]. The reduced fraction will be left in Y and X with the new numerator in Y and the new denominator in X.

5) To add, subtract, multiply, or divide any two fractions place the first (top) fraction in the T & Z registers with the numerator in T and the denominator in Z. Place the second (bottom) fraction in Y & X with the numerator in Y and the denominator in X. For these four operations the calculator assumes:

T/Z + Y/X   T/Z - Y/X   T/Z * Y/X   T/Z / Y/X

Press the key corresponding to the desired operation. The answer will be left with the numerator in Y and the denominator in X.

MORE EXAMPLES OF FR

Example 3: Reduce 273/924

Set F10 to see the answer displayed. Key 273 ENTER↑ 924 and push [REDUCE]. See 13/44 returned.

Example 4: Subtract 7/8 - 5/18

Key 7 ENTER↑ 8 ENTER↑ 5 ENTER↑ 18. The stack will be loaded as:

```
T:  7
Z:  8
Y:  5
X: 18
```

Push [-] and see 43/72 displayed. The final stack contains:

```
T: *
Z: *
Y: 43
X: 72
```

Example 5: Calculate (13/40)/(26/35)

Key 13 ENTER↑ 40 ENTER↑ 26 ENTER↑ 35. The stack will be loaded as:

```
T: 13
Z: 40
Y: 26
X: 35
```

Press [/] and see 7/16. The final stack contains:

```
T: *
Z: *
Y:  7
X: 16
```

FR

Example 6: Calculate ((4/25)/(3/5))*(5/6)

Key 4 ENTER↑ 25 ENTER↑ 3 ENTER↑ 5 and then push [/]. See 4/15. After dividing the stack contains the partial result:

```
T: *
Z: *
Y:  4
X: 15
```

To continue simply key 5 ENTER↑ 6 so the stack will contain:

```
T:  4
Z: 15
Y:  5
X:  6
```

before multiplying. Push [*] and see 2/9 as the final answer.

FORMULAS USED IN FR

As indicated previously, the 4 basic operations are carried out in the stack with the first fraction assumed to be in the T and Z registers and the second fraction assumed to be in the Y and X registers. The resulting fraction is left in Y and X in reduced form.

(1) T/Z + Y/X = (XT + YZ)/XZ

(2) T/Z - Y/X = (XT - YZ)/XZ

(3) T/Z * Y/X = (TY)/(XZ)

(4) T/Z / Y/X = (TX)/(YZ)

In addition the greatest common divisor routine uses the iteration formula:

(5) $R_{i+1} = R_{i-1}$ MOD R_i so that GCD = R_i when $R_{i+1}=0$.

Routine Listing For: FR

```
01♦LBL "FR"        25♦LBL 05
02 GTO IND 06      26 RCL Y
03♦LBL B           27 RCL Y
04♦LBL 02          28 XEQ 06
05 CHS             29 ST/ Z
06♦LBL A           30 /
07♦LBL 01          31 FC? 10
08 ST* T           32 RTN
09 X<> Z           33 FIX 0
10 *               34 " "
11 ST+ Z           35 ARCL Y
12 X<> L           36 "├/"
13 *               37 ARCL X
14 GTO 05          38 XROM "VA"
15♦LBL D           39 RTN
16♦LBL 04          40♦LBL c
17 X<>Y            41♦LBL 06
18♦LBL C           42 MOD
19♦LBL 03          43 LASTX
20 ST* Z           44 X<>Y
21 X<> T           45 X≠0?
22 *               46 GTO c
23 X<>Y            47 +
24♦LBL E           48 RTN
```

LINE BY LINE ANALYSIS OF FR

Line 02 provides access to all the subroutines within FR.

Lines 03-05 are used to set up the subtraction routine which feeds directly into the addition routine after negating the fraction to be subtracted.

Lines 06-14 are the addition routine.

Lines 15-17 are used to set up the division routine which feeds directly into the multiplication routine after inverting the fraction that is the divisor.

Lines 18-23 are the multiplication routine.

Lines 24-39 are the reducing routine which ends at line 32 unless flag 10 has been set in which case the resulting fraction is shown in the alpha register.

Lines 40-48 are the greatest common divisor routine which is called by the reducing routine.

NUMERIC LABELS/FUNCTIONS IN THE FR PROGRAM

The following list gives a correspondence between numeric labels and subroutines to be called as part of FR programs. To call a subroutine function from one of your own programs, first store the number corresponding to the desired function in data register R06. Then use the instruction XEQ "FR" as part of your program.

Numeric Label Number in R06	Keyboard Label	Subroutine Function
01	A	Addition of fractions
02	B	Subtraction of fractions
03	C	Multiplication of frac.
04	D	Division of fractions
05	E	Reduction of fractions
06	c	Greatest Common Divisor of X and Y.

In addition to the intended use of the above numeric subroutines, access is also available to the numeric labels in the DF and NP routines. For the truly curious PPC member this access may provide hidden functions in the PPC ROM.

CONTRIBUTORS HISTORY FOR FR

The FR routine and documentation were written by John Kennedy (918). John Dearing (2791) suggested improvements.

FINAL REMARKS FOR FR

All calculators should provide the basic functions in FR.

FURTHER ASSISTANCE ON FR

John Kennedy (918) phone: (213) 472-3110 evenings
Ron Knapp (618) phone: (213) 867-3086

NOTES

TECHNICAL DETAILS

XROM: 20, 12	FR	SIZE: 000

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: set=display fraction; clear=no display
- 25: used in VA routine

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

alpha registers used only if flag F10 is set to display fractions

Other Status Registers:
- 9 Q: not used
- 10 ⊦: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

Display Mode:
FIX 0

recommend F29 clear

Angular Mode:
not used

Unused Subroutine Levels:
4

ΣREG: not used

Data Registers:

R00:

R06:

R07:

R08:

R09:

R10:

R11:

R12:

FR does not use any data registers all computations are carried out in the stack

Global Labels Called:

Direct	Secondary
VA if F10 set	none

Local Labels In This Routine:

A, B, C, D, E, c

01, 02, 03, 04, 05, 06

Execution Time: See NUMERIC LABELS section in the FR documentation.

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: FR

Bytes In RAM: 84

Registers To Copy: 36

Other Comments:

CONTINUED FROM PAGE 169

MEM	QUAN	ORD
6148	1	989
6149	1	2150
6151	2	1345
		1787
6152	1	1206
6153	1	1546
6155	1	1288
6156	1	1053
6157	1	986
6158	1	1117
6161	1	1767
6165	1	1066
6166	1	1146
6167	1	1260
6168	1	1406
6169	1	1274
6170	2	1194
		1195
6171	2	1259
		2028
6172	1	1427
6173	1	1818
6174	1	1426
6175	1	1793
6176	1	1425
6177	1	1136
6178	1	1418
6180	1	1349
6181	2	1336
		2085
6186	3	1417
		1679
		1680
6191	1	1332
6192	1	2358
6194	1	1859
6197	1	1350
6198	3	1340
		1341
		1342
6200	1	2205
6201	1	2470
6202	1	2179
6203	1	1070
6204	1	1331
6205	1	2048
6208	1	1351
6210	2	1613
		1614
6211	1	1423
6212	3	1920
		1921
		1922
6213	1	1744
6215	1	1777
6219	1	1622
6221	1	1729
6223	1	2215
6224	1	1416
6226	1	1645
6227	3	1410
		1666
		1667
6228	1	1424
6229	2	1414
		1415
6230	1	2307
6231	1	1374
6233	2	1697
		1698
6234	1	1409
6235	1	1707
6236	1	1689

MEM	QUAN	ORD
6237	1	1373
6238	1	1376
6240	2	1702
		2107
6241	1	1408
6242	1	1983
6243	1	1629
6245	1	1539
6246	1	1378
6247	1	1525
6250	2	1381
		1382
6255	1	1789
6257	1	2171
6259	1	2096
6262	1	1377
6263	1	1375
6264	1	1663
6268	1	1703
6270	1	1604
6271	1	1951
6273	1	1529
6274	1	1770
6276	1	1834
6278	2	1827
		1828
6279	1	1609
6280	1	1608
6281	1	2156
6282	1	2239
6283	1	1630
6284	1	1804
6285	1	1950
6287	1	2230
6293	1	1960
6295	1	1819
6299	2	1995
		1996
6302	1	1753
6307	2	1992
		1993
6308	1	1706
6309	1	1786
6310	1	2104
6312	1	2176
6316	1	1626
6317	1	1634
6319	1	1633
6322	1	2430
6324	1	1632
6325	1	1694
6327	1	1692
6328	1	1690
6330	1	1825
6331	1	2298
6335	1	1846
6336	1	2025
6338	2	1758
		1759
6339	1	2175
6344	1	1708
6345	3	1938
		1939
		1940
6346	1	1669
6351	1	1959
6353	1	1756
6355	1	1783
6356	1	1907
6358	1	1805
6359	1	1911
6365	1	1745
6366	1	1668

MEM	QUAN	ORD
6368	1	1824
6373	1	2320
6374	1	2128
6376	1	1665
6377	1	1705
6379	1	1782
6380	1	1664
6382	1	1248
6383	3	1673
		1674
		1675
6384	2	1671
		1672
6386	3	1670
		2124
		2125
6391	1	1814
6393	1	2332
6397	1	1687
6398	1	1749
6400	1	1847
6402	1	2146
6403	3	1737
		1738
		1739
6405	1	1989
6411	1	1826
6412	1	2004
6417	1	1792
6418	1	2010
6420	1	1823
6421	1	2201
6423	1	1791
6424	1	1811
6426	1	1727
6427	1	1925
6428	1	2187
6429	1	1941
6430	1	2378
6435	1	1835
6436	2	2064
		2065
6437	1	2333
6439	1	2024
6440	1	1836
6441	1	2160
6442	3	2221
		2222
		2223
6444	1	2006
6451	1	1970
6454	1	1837
6458	1	2177
6459	1	2083
6460	1	2390
6462	1	1775
6464	1	2173
6465	1	2341
6466	1	1865
6467	1	2016
6470	1	2434
6472	1	2080
6473	1	2021
6475	1	1838
6477	1	1987
6481	1	1839
6486	1	2268
6487	1	2181
6490	1	2353
6495	3	2232
		2233
		2234
6498	1	2212

MEM	QUAN	ORD
6502	1	2306
6504	1	1909
6510	1	1889
6513	1	1998
6515	1	2099
6516	2	1968
		1969
6517	1	1899
6518	1	2139
6520	1	2130
6523	1	2180
6524	1	1883
6528	1	2235
6529	1	2086
6530	1	2076
6533	1	2003
6537	2	1985
		1986
6539	1	1905
6541	1	2406
6542	1	2149
6545	1	2140
6547	1	2340
6550	1	1852
6551	1	2227
6552	4	1931
		1932
		1933
		1934
6554	1	2111
6555	1	2077
6556	1	2194
6557	1	2258
6558	3	2314
		2493
		2494
6560	1	2105
6561	1	1928
6563	1	2186
6564	1	1929
6565	1	2241
6566	2	2013
		2014
6569	2	2209
		2210
6573	1	1930
6574	2	1881
		1882
6575	1	1935
6578	1	2095
6582	1	1868
6584	1	2153
6585	1	1875
6587	3	1913
		1914
		1915
6591	1	1943
6592	1	2189
6593	1	2392
6594	1	2454
6598	1	2054
6601	1	2094
6604	1	1916
6606	1	1944
6607	1	2159
6608	2	2121
		2122
6610	1	1927
6614	1	1961
6615	1	1957
6616	2	1107
		2166

MEM	QUAN	ORD
6618	1	2148
6620	1	1902
6621	1	1962
6623	1	2103
6632	1	2188
6633	1	2019
6635	1	2238
6637	1	1918
6638	1	1980
6639	1	2090
6640	1	1977
6642	1	1976
6646	1	2248
6647	1	2151
6650	1	2397
6652	1	2391
6655	1	1964
6658	1	1870
6660	1	1891
6662	1	2240
6663	1	1896
6664	1	2313
6668	2	1871
		1872
6669	1	2228
6670	1	1879
6675	1	2287
6676	1	2423
6679	1	1967
6680	1	2243
6681	1	1880
6686	1	2237
6687	1	2299
6689	1	1966
6690	1	1041
6694	1	1956
6696	1	2271
6698	1	2068
6704	1	2373
6706	1	2374
6708	1	1954
6712	1	2075
6713	1	2074
6714	1	2073
6715	1	2042
6716	1	2072
6717	1	2071
6718	1	2063
6719	1	2041
6720	1	2040
6721	1	2038
6722	1	2039
6723	1	2037
6724	3	2034
		2035
		2036
6725	1	2032
6726	1	2033
6727	1	2031
6728	1	2030
6729	1	2027
6730	1	2047
6731	1	2292
6732	1	2060
6733	1	2055
6734	1	2058
6735	1	2242
6736	1	2245
6737	1	2309
6738	1	2246
6739	1	2247
6740	1	2270
6741	1	2272

MEM	QUAN	ORD
6742	1	2273
6743	1	2274
6744	1	2281
6746	1	2479
6749	1	2300
6750	1	2396
6756	1	2285
6757	2	2283
		2284
6760	1	2293
6761	1	2290
6763	1	2408
6765	1	2448
6766	1	2178
6773	1	2375
6775	1	2305
6777	1	2443
6778	1	2364
6794	1	2435
6805	1	2291
6811	1	2346
6821	1	2449
6823	1	2469
6826	1	2365
6828	1	2442
6834	1	2416
6836	1	2376
6838	1	2417
6839	1	2318
6848	2	2288
		2357
6853	1	2289
6854	1	2432
6858	1	2326
6859	1	2412
6862	1	2462
6866	1	2369
6878	1	2295
6880	1	2294
6886	3	2303
		2490
		2498
6887	3	2304
		2496
		2497
6890	1	2377
6891	1	2425
6901	1	2360
6903	1	2458
6916	1	2363
6918	1	2336
6928	1	2398
6938	1	2447
6940	1	2361
6946	1	2371
6948	1	2404
6950	1	2402
6951	1	2452
6952	1	2386
6958	1	2389
6959	1	2471
6976	1	2334
6979	1	2362
6981	1	2478
6986	1	2330
6992	1	2424
6999	1	2387
7009	1	2338
7010	1	2468
7012	1	2400
7014	1	2370

MEM	QUAN	ORD
7019	1	2441
7021	1	2354
7023	1	2372
7025	1	2382
7030	1	2345
7033	1	2465
7038	1	2383
7042	1	2351
7048	1	2315
7049	1	2393
7061	1	2328
7065	1	2324
7066	1	2440
7070	1	2415
7072	1	2453
7074	1	2312
7084	1	2477
7096	1	2421
7097	1	2405
7102	1	2327
7105	1	2331
7112	1	2403
7113	1	2359
7114	1	2329
7145	1	2410
7146	1	2381
7153	1	2399
7162	1	2368
7182	1	2409
7185	1	2380
7190	1	2342
7191	1	2457
7194	1	2413
7204	1	2422
7205	1	2325
7210	1	2459
7223	1	2476
7231	1	2067
7238	1	2445
7242	1	2444
7243	1	2414
7245	1	2467
7286	1	1369
7308	1	2455
7309	1	2456
7310	1	2451
7777	8	2475
		2481
		2482
		2484
		2489
		2491
		2499
		2500

END

GE - GO TO .END.

GE provides a quick way out of ROM, leaving the program pointer at line 00 of the last program file. This is especially handy when you're using the last program file as a scratch area for trying things out. Assigning GE and DEL or CLP to keys will prove helpful in keying up the Examples in this manual. Many of the examples using keyboard operations will leave you in ROM, and GE can get you back to your program example scratch area.

Example 1: XEQ GE and hold down the PRGM mode switch until the annunciator turns on. SST to verify that you are now at line 00 of the last program file (the one that has .END. instead of END). Notice that you got there without packing or adding an END, as GTO.. would have done.

GE can also be used in a program (for instance, EP, PA) to bring the program pointer back to a known RAM location, to eliminate unwanted return levels, or to reset the line number to zero in preparation for STO b. The former characteristic is of no use in user RAM programs, but the last two may be.

GE works by replacing the subroutine return stack with a single return pointer to the .END.. The .END. is then executed, placing the pointer at line 00. The pointer to the .END. had to be put in the return stack for activation by RTN, because STO b behaves differently in ROM than in RAM. See the Ab write-up for details.

COMPLETE INSTRUCTIONS FOR GE

Just XEQ GE from the keyboard or in a program. No inputs are required. In about one second execution will halt with the program pointer at line 00 of the last user program file. Execution halts regardless if any returns that were pending at the time GE was called. The stack contents, including L, are preserved, except that T and alpha are cleared.

MORE EXAMPLES OF GE

Example 2: You have read in (and perhaps executed) an HP-67 program, and have then gone elsewhere in RAM or ROM to execute another program. To get back to the program (which has no global label) you would normally have wait through all of catalog 1. With the PPC ROM you just XEQ GE and you're there. This assumes that you have not done GTO.. or otherwise put an END on your HP-67 program.

LINE BY LINE ANALYSIS OF GE

Line 160 recalls the current program pointer for replacement of the return address portion. Lines 159 and 162 use flag 25 to determine whether line 161 is being executed for the first or second time. The second time through (immediately after 182 ASTO b) we are ready to RTN because the return pointer has been set to the .END. location. This setting of the return pointer is done by lines 164-180.

Line 164 puts the program pointer from line 160 in the N register, after which line 166 calls a subroutine which isolates the .END. pointer from C. Lines 183 to 186 place c in M and shift it 5 bytes left. This puts the last two bytes of the stored program pointer in the first two bytes of N and the last two bytes of c in the first two bytes of M. Lines 187-194 pull out these last two bytes of c and clear the first nybble. What remains is three hexadecimal digits

GE

TECHNICAL DETAILS

XROM: 10,60 — GE — SIZE: 000

Stack Usage:
- 0 T: zero
- 1 Z: UNCHANGED
- 2 Y: UNCHANGED
- 3 X: UNCHANGED
- 4 L: UNCHANGED

Flag Usage: SEVERAL USED BUT ALL RESTORED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10:
- 25:

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

ALL CLEARED

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: CLEARED
- 12 b: BYTES 1-5 CLEARED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 0

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R06:
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine:

NONE

Execution Time: 1.1 second.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	YES
Program File:	IF
Bytes In RAM:	99
Registers To Copy:	60

Other Comments:

(second through fourth nybbles) that constitute the .END. pointer. Lines 167 to 170 set the two bits immediately to the left of the .END. pointer. At this point the first two bytes of X contain a program pointer in RAM return format (see RT for an explanation of return pointer format). This return pointer corresponds to byte 3 of the register containing the .END. Since bytes are numbered 6 to 0 in line number order, the .END. instruction itself occupies bytes 2, 1, and 0 of its register. A program pointer causes execution to begin at the following byte, so when this return pointer is activated by RTN the .END. will be the first instruction executed. But first we have to get the return pointer into the b register.

Lines 171-173 put the return pointer in the first two bytes of M and pull the line 160 program pointer out of N. Line 174 shifts the return pointer into the last two bytes of N, then line 175 stores the program pointer right next to it in the first two bytes of M. This combination is then shifted into the rightmost form bytes of N, extracted while alpha is cleared, and put back in M. It is now ready for ASTO b, which jumps control up to line 161 (immediately following the place where RCL b was executed). Alpha is cleared and the RTN on line 163 jumps control to the .END.. When the .END. is encountered execution stops at line 00 of the last program file in RAM.

CONTRIBUTORS HISTORY FOR GE

GE was conceived by Roger Hill (4940). The ROM version was written by Roger Hill and Keith Jarett (4360).

FURTHER ASSISTANCE ON GE

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

Routine Listing For: **GE**

```
158♦LBL "GE"        178 CLA
159 SF 25           179 X<> [
160 RCL b           180 RDN
161 CLA
162 FC?C 25         181♦LBL "Ab"
163 RTN             182 ASTO b
164 STO \
165 RDN             183♦LBL 14
166 XEQ 14          184 RCL c
167 X<> d           185 STO [
168 SF 05           186 "├♦♦♦♦♦♦"
169 SF 06           187 X<> [
170 X<> d           188 X<> d
171 STO [           189 CF 00
172 CLX             190 CF 01
173 X<> \           191 CF 02
174 "├**"           192 CF 03
175 STO [           193 X<> d
176 "├**"           194 RTN
177 X<> \
```

NOTES

GN - GAUSSIAN RN GENERATOR

This routine is a special random number generator which produces a Gaussian (bell-shaped) distribution (also called normal distribution) where the mean and standard deviation are specified by the user. This routine calls the RN routine and hence requires a register pointer in X when used. See also the RN routine. GN returns two numbers in the specified range. GN also uses the rectangular-polar coordinate functions and should be used in Degrees mode.

Example 1: Use GN to produce a series of random numbers centered around the number 50 where the standard deviation is 5.

First make sure the HP-41C is in Degrees mode. As in the examples using RN, the GN routine requires the use of one data register to hold the initial and subsequent random number decimals r in the range 0<r<1. We arbitrarily choose register R05 to hold the seeds and choose the fractional part of pi as the initial seed. Key .141592654 STO 05. The RN routine is now ready.

To initialize GN store the desired mean (in this example 50) in R06. 50 STO 06. Store the standard deviation (in this example 5) in R07. 5 STO 07. In a standard normal curve approximately 68% of the data lies within one standard deviation of the mean, 95% of the data is within two standard deviations of the mean, and 99.7% of the data is within three standard deviations of the mean.

This means 68% of the numbers produced by GN in this example should fall in the range 45-55. Almost all (95%) of the numbers will be in the range 40-60. 99.7% of the numbers will be in the range 35-65 and it would be very rare for GN to produce a number outside the 35-65 range in this example.

GN produces two numbers each time it is called. The following values are given to 2 decimal places.

Do:	Result: X	Y
5 XEQ "GN"	52.43	52.38
5 XEQ "GN"	45.29	51.24
5 XEQ "GN"	61.54	52.43
5 XEQ "GN"	47.80	44.28
5 XEQ "GN"	56.73	47.18
5 XEQ "GN"	51.87	53.28
5 XEQ "GN"	43.09	49.20
5 XEQ "GN"	47.15	58.96
5 XEQ "GN"	55.22	51.21
5 XEQ "GN"	55.66	54.83

Note that the values returned mostly lie in the 45-55 range. From the 10 pairs generated so far the low is 43.09 and the high is 61.54. 13 of the 20 numbers (65%) are in the 45-55 range, 19 of the 20 (95%) are in the 40-60 range, and all 20 (100%) are in the 35-65 range.

GN

COMPLETE INSTRUCTIONS FOR GN

1. One data register is required to hold the seeds for the RN routine which GN calls. The number (address) of the data register is the input to GN each time GN is called. Call this register k. Before the first call to GN store any decimal between 0 and 1 in register k.

2. Store the desired mean in R06.

3. Store the desired standard deviation in R07.

Note that the mean and standard deviation together determine the range of numbers produced by GN. 68% of the numbers will be within one standard deviation of the mean. 95% of the numbers will be within two standard deviations of the mean. 99.7% of the numbers produced by GN will be within three standard deviations of the mean.

4. Set the angle mode to Degrees.

5. To generate the next number key in k and XEQ "GN".

6. Step 5 may be repeated any number of times.

7. The stack register contents on input/output to GN are indicated by:

Input:		Output:	
	T: T		T: k (pointer)
	Z: Z		Z: k (pointer)
	Y: Y		Y: 1st number
	X: k (pointer)		X: 2nd number
	L: L		L: mean (R06)

MORE EXAMPLES OF GN

Example 2: Use GN to generate a sequence of numbers whose mean is 500 and whose standard deviation is 25.

Insure the HP-41C is in Degrees mode. Store 500 in R06 and store 25 in R07. Use register R05 for the seeds and initialize R05 with the fractional part of e. .718281828 STO 05.

The following pairs of numbers should be produced by successively keying in 5 and XEQ "GN".

X	Y
531.03	496.02
488.55	516.86
527.79	549.94
498.45	487.20
501.45	539.30
497.29	508.39
540.58	471.95
513.81	512.77

The one-standard-deviation range is 475-525.
The two-standard-deviation range is 450-550.
The three-standard-deviation range is 425-575.

Example 3: Continue Example 2 without entering a new initial seed. Keep the same mean but change the standard deviation to 100. Key 100 STO 07. The following pairs will be generated by successively keying in 5 and XEQ "GN".

X	Y
466.99	595.12
606.52	466.36
660.18	678.56
395.47	475.60
646.40	609.68
479.42	666.86

FORMULAS USED IN GN

(1) T_1 = SQRT(-2*ln(R_1)*sin(360*R_2)

(2) T_2 = SQRT(-2*ln(R_1)*cos(360*R_2)

where the trigonometric functions are used in degrees mode and and R_1 and R_2 are the random numbers produced by RN.

(3) N_1 = s*T_1 + m

(4) N_2 = s*T_2 + m

where s and m are the user specified standard deviation and mean respectively and N_1 and N_2 are the final numbers produced by GN.

Routine Listing For: GN

```
126♦LBL a
127♦LBL "GN"
128 XEQ b
129 LN
130 ST+ X
131 CHS
132 SQRT
133 X<>Y
134 XEQ b
135 360
136 *
137 R↑
138 RCL 07
139 *
140 P-R
141 RCL 06
142 ST+ Z
143 +
144 RTN
```

LINE BY LINE ANALYSIS OF GN

Lines 128 and 134 are the calls to the RN routine.

Line 140 takes advantage of the P-R function to calculate the sine and cosine in one step.

Lines 138 and 139 scale the values for the user specified standard deviation.

Lines 141-143 translate the values for the user specified mean.

REFERENCES FOR GN

Kiyoshi Akima (3456) "PPC Journal" Two RN's For The Price of One V6N3P8 V6N6P2 NOP COLUMN

See also the references for RN, in particular see Knuth, Semi-Numerical Algorithms p. 104

CONTRIBUTORS HISTORY FOR GN

The technique used to generate a bell-shaped distribution from a uniform one was first suggested by Kiyoshi Akima (3456) in V6N3P8. John Kennedy (918) adapted this method for the HP-41C to be used in degrees mode. John also wrote the documentation for GN.

FURTHER ASSISTANCE ON GN

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

T E C H N I C A L   D E T A I L S

XROM: 20, 15 — GN — SIZE: 001 minimum

Stack Usage:
0 T: used
1 Z: used
2 Y: used
3 X: used
4 L: used

Alpha Register Usage:
5 M: not used
6 N: not used
7 O: not used
8 P: not used

Other Status Registers:
9 Q: not used
10 ⊢: not used
11 a: not used
12 b: not used
13 c: not used
14 d: not used
15 e: not used

ΣREG: not used

Data Registers:
R00: GN requires one RAM register to hold the seeds for RN.
R06: =user specified mean
R07: =standard deviation
R08: not used
R09: not used
R10: not used
R11: not used
R12: not used

Flag Usage:
04: not used
05: not used
06: not used
07: not used
08: not used
09: not used
10: not used
25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
RN	none

Local Labels In This Routine:
a

Execution Time: 2.9 seconds

Peripherals Required: none

Interruptible? yes
Execute Anytime? no
Program File: FR
Bytes In RAM: 36
Registers To Copy: 36

Other Comments:

HA - HIGH RESOLUTION HISTOGRAM WITH AXIS

This routine will generate in the print buffer, a high resolution bar-chart bar, extending from a user-prescribed axis position, for use in charts or histograms. The height of the bar may be from 1 to 168 printer columns, with the axis position in any column. If the value to be plotted is greater than the axis value, then the bar is plotted up (from left to right) from the axis to the value. If the value is lower than the axis, the bar is plotted up from the value to the axis. If the value and the axis occupy the same column, then no symbol is printed.

The user specifies the fill-character from the standard printer character set, to fill in 7-column sections of the bar. The remaining printer columns are filled by a user-defined fill-column using printer function ACCOL values 0 (no dots filled) to 127 (all dots filled). Bars created by HA are accumulated in the print buffer but not printed, thus allowing information to be added later. The inputs to HA match those of the printer's REGPLOT routine, allowing free substitution between the two. A table of the standard printer character set appears on the last page of the writeup for routine HS

Example 1. Plot a sine curve using the HA routine. X limits are 0 to 360 degrees, with an increment of 18 degrees per line. Y limits are to be -1 to +1, with a plot width of the full 168 columns and an axis at value 0 (column 84). Use printer symbol #125 (the right facing arrow) for the fill character, and ACCOL #8 (middle dot 'turned on') for the fill column.

The following program does the job:

APPLICATION PROGRAM FOR: HA	
01♦LBL "SINE"	
02 CF 12	Set lower case
03 "-1"	
04 ACA	
05 66	Accumulation
06 SKPCOL	of
07 "0"	Y directional
08 ACA	numeric labels
09 74	
10 SKPCOL	
11 "1"	
12 ACA	
13 PRBUF	
14 SF 12	
15 "------"	
16 ACA	Accumulation
17 CF 12	of
18 "-"	Y axis
19 ACA	into print
20 6	buffer
21 SKPCOL	
22 127	
23 ACCOL	
24 SF 12	
25 "-------"	
26 ACA	
27 PRBUF	
28 CF 12	
29 -1	
30 STO 00	Y minimum
31 CHS	
32 STO 01	Y maximum
33 168.084	
34 STO 02	Plot width/Axis col.
35 125	
36 STO 03	Fill character
37 8	
38 STO 05	Fill column
39 .36018	
40 STO 06	Numeric counter
41♦LBL 00	
42 RCL 06	
43 INT	
44 SIN	
45 XROM "HA"	Call to HA
46 PRBUF	Print the buffer
47 ISG 06	
48 GTO 00	
49 END	

The printer output appears in figure 1.

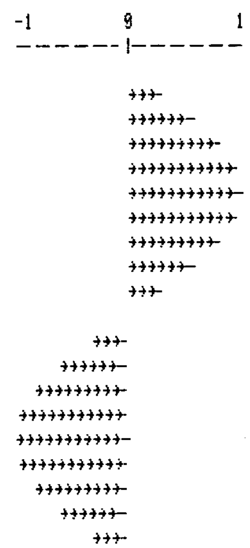


Figure 1. Plotted output of sine curve from example 1.

Example 2. Plot the values in table 1 below, using HA, with the X-axis labelled by its value in FIX 0 display mode. Position an axis in the bar chart at value 150.

X	Y
100	100
150	175
200	225
250	240
300	275
350	260
400	140
450	90
500	180

Table 1. Values to be plotted in example 2.

Let us choose our Y minimum and maximum to be 0 and 300, respectively, and choose the fill character to be printer symbol #31 (the symbol closest to filling the full 7 by 7 block). The fill column shall be ACCOL #127 (all dots in the column filled) for a "flat top" to the bars. Labelling the chart with the X values requires 4 printer character positions (3 for the number plus one for a space) or 28 columns. This leaves 140 columns for the bar chart itself. With an axis at 150, or exactly halfway across the field of 0 to 300, we place the axis in column 70. The resulting keystroke sequence is:

Keystrokes	Display	Result
FIX 0	X	Display mode
0	0	
STO 00	0	Y minimum
300	300	
STO 01	300	Y maximum
140.070	140.070	
STO 02	140	Plot width . Axis position
31	31	
STO 03	31	Fill character
127	127	
STO 05	127	Fill column
100	100	
ACX	100	1st X label
XEQ HA	127.0000	First bar
PRBUF	127.0000	Print buffer
FIX 0	X	Reset display
150	150	
ACX	150	2nd X label
175	175	
XEQ HA	127.0000	Second bar
PRBUF	127.0000	Print buffer
⋮	⋮	⋮
FIX 0	X	Reset display
500	500	
ACX	500	Last X label
180	180	
XEQ HA	127.0000	Last bar
PRBUF	127.0000	Print buffer

The resulting bar graph is shown in figure 2.

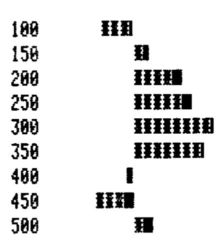


Figure 2. Bar graph using HA from data in table 1, example 2.

COMPLETE INSTRUCTIONS FOR HA

Fill in the necessary data registers with the required information as follows:

Register	Contents
R00	Y minimum
R01	Y maximum
R02	nnn.aaa where: nnn = plot width in columns (less than or equal to 168) aaa = column position of axis
R03	Fill character (#0 to #127 printer char.)
R05	Fill character (#0 to #127 ACCOL number)
X register	Value to be plotted

Then XEQ HA and a bar will be accumulated into the print buffer. The information in the data registers coincides with that used in the printer routine REGPLOT. The HA routine leaves the display mode in FIX 4, so if another mode is desired, it must be set after execution of HA .

MORE EXAMPLES OF HA

Example 3. Plot a bar chart of the number of pages in each issue of the PPC Calculator Journal in 1980, with 30 pages as an axis position. Use 20 to 64 as Y limits.

Let us choose our fill character to be printer symbol 35 (#) and our fill column to be ACCOL #20 (3rd and 5th columns filled). The data is shown in table 2.

Issue No.	No. of Pages
1	32
2	56
3	32
4	32
5	64
6	48
7	32
8	32
9	32
10	24

Table 2. Page length of the ten 1980 issues of the PPC Calculator Journal.

Let us label the left hand edge of our chart with the issue numbers, requiring two printer characters, plus an extra two spaces to separate the labels from the chart. Therefore 4 characters or 28 printer columns will not be available for plotting the bars. A total of 168-28 or 140 columns will remain. In order to have an axis at position in a range of 20 to 64, we calculate the column to be:

140	*	(30-20)	/	(64-20)	=	31.8
columns of plot width		axis-bottom		top-bottom		axis position

We can round this number to column 32, in order to load register 02 with the number 140.032. The keystroke sequence for this bar graph would then be:

Keystrokes	Display	Result
20	20	
STO 00	20.0000	Y minimum
64	64	
STO 01	64.0000	Y maximum
140.032	140.032	
STO 02	140.0320	Plot . Axis width position
35	35	
STO 03	35.0000	Fill character
20	20	
STO 05	20.0000	Fill column
ALPHA (space) 1 (space) (space) ALPHA	1	X label
ACA	20.0000	Load buffer
32	32	
XEQ HA	20.0000	First bar
PRBUF	20.0000	Print buffer
ALPHA (space) 2 (space) (space) ALPHA	2	X label
ACA	20.0000	Load buffer
56	56	
XEQ HA	20.0000	Second bar
PRBUF	20.0000	Print buffer
.	.	.
.	.	.
.	.	.
ALPHA 1 0 (space) (space) ALPHA	10	X label
ACA	20.0000	Load buffer
24	24	
XEQ HA	20.0000	Last bar
PRBUF	20.0000	Print buffer

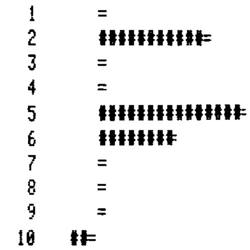


Figure 3. Bar graph of PPC Calculator Journal lengths for the year 1980, produced by the HA routine.

Routine Listing For: **HA**

```
12♦LBL "HA"
13 RCL 01
14 X>Y?
15 X<>Y
16 RCL 00
17 -
18 RCL 01
19 RCL 00
20 -
21 /
22 X<0?
23 .
24 RCL 02
25 INT
26 ST* Y
27 X<>Y
28 FIX 4
29 RND
30 X<>Y
31 RCL 02
32 FRC
33  E3
34 *
35 X<=Y?
36 X<>Y
37 RDN
38 X>Y?
39 X<>Y
40 STO Z
41 -
42 STO 04
43 RDN
44  E
45 -
46 SKPCOL
47  E
48♦LBL "HS"
49 RCL 04
50 *
51 LASTX
52 X>Y?
53 X<>Y
54 INT
55 7 E-5
56 +
57 RCL 03
58 GTO 00
59♦LBL 01
60 ACCHR
61♦LBL 00
62 DSE Y
63 GTO 01
64 RDN
65 INT
66 8
67 +
68 RCL 05
69 GTO 00
70♦LBL 02
71 ACCOL
72♦LBL 00
73 DSE Y
74 GTO 02
75 RTN
```

LINE BY LINE ANALYSIS OF HA

Lines 12 through 15 check if the value to plotted exceeds Y maximum.

Lines 16 through 26 scale the value into a number of columns and corrects if the value is negative.

Lines 27 through 30 rounds the column number.

Lines 31 through 36 compare the value's column number to the column number of the axis.

Lines 37 through 42 adjust the position of the value if it falls below the axis position, and places a new 'plot width' value into register 04.

Lines 43 through 47 inserts a SKPCOL to the beginning of the bar to be added to the buffer. Routine HS (in lines 48 through 75) takes care of adding the bar to the print buffer.

TECHNICAL DETAILS

XROM: 20,25	HA	SIZE: 006

Stack Usage: 0 T: 1 Z: ALL USED 2 Y: 3 X: 4 L:	**Flag Usage:** NONE USED 04: 05: 06: 07: 08: 09: 10: 12: CLEAR 13: CLEAR 25:
Alpha Register Usage: 5 M: 6 N: NONE USED 7 O: 8 P:	
Other Status Registers: 9 Q: 10 ⊦: 11 a: NONE USED 12 b: 13 c: 14 d: 15 e:	**Display Mode:** Routine returns in FIX 4 **Angular Mode:** NOT USED **Unused Subroutine Levels:** 5
ΣREG: NOT USED **Data Registers:** R00: Y MINIMUM R01: Y MAXIMUM R02: PLOT WIDTH/AXIS POS. R03: FILL CHARACTER R04: USED INTERNALLY R05: FILL COLUMN R10: NOT USED R11: NOT USED R12: NOT USED	**Global Labels Called:** Direct / Secondary No labels are called, however HA continues executing in HS. **Local Labels In This Routine:** NONE
Execution Time: 4 to 6 seconds.	
Peripherals Required: 82143A Printer	
Interruptible? YES Execute Anytime? NO Program File: LG Bytes In RAM: 50 Registers To Copy: 29	**Other Comments:** This routine loads the print buffer, but does not PRBUF. Flags 12 and 13 must be clear to run HA.

REFERENCES FOR HA

See PPC Calculator Journal, V7N10P11b.

CONTRIBUTORS HISTORY FOR HA

This routine was an out growth of the work of Ron Gordon on the original high resolution histogram program without an axis (see HS). The following individuals either were helpful in providing useful suggestions or aided in the debugging process: Wayne Beimesch (5854), Cliff Carrie (834), Bill Hermanson (4115), Geoff Kuenning (5071) and Steve Wandzura (4635).

FINAL REMARKS FOR HA

Routine HA is an outgrowth of HS, the histogram routine without axis. It is suggested that the user familiarize himself with both of these routines together, in order to better understand the capabilities of histogram and bar-chart plotting using the PPC ROM.

FURTHER ASSISTANCE ON HA

Contact Jake Schwartz (1820) at 7700 Fairfield St., Phila., Penna. 19152 (home phone 215-331-5324); or Cliff Carrie (834) at 152 Beverley Ave., Mount Royal, Quebec, Canada H3P1K7 (home phone 514-733-4866).

NOTES

HD - HIDE DATA REGISTERS

This routine is used to raise the curtain, and to prepare for a rapid lowering of the curtain via UD. It uses a single data register (R00, after the curtain is raised) to save the last 5 bytes of status register c; so the contents of this register must not be changed before UD uses it or MEMORY LOST will result.

BACKGROUND FOR HD

See Appendix M on Curtain Moving.

Example 1: The sequence 6, XEQ HD places status register c data into R06, then raises the curtain by 6 registers (so that R06 becomes R00).

COMPLETE INSTRUCTIONS FOR HD

The complementary pair of routines HD and UD provide the fastest means for the type of repeated curtain manipulation required to call within a computing loop a routine which otherwise would destroy data needed by the calling routine. The important constraint is that, after raising the curtain, the routine called does not change the contents of the new R00, so that UD can be used to lower the curtain very quickly to its former position. The calling scheme is:

```
  :
  n
XROM HD
XEQ subroutine
XROM UD
```

HD requires SIZE ≥ n + 6, because it sets ΣREG to n. HD preserves the former Y, Z, and T in X, Y, and Z. This makes it easy to provide subroutine inputs in the stack. In particular the XROM HD and XROM UD instructions can be put in the subroutine itself.

The HD / UD pair are readily used for nested curtain manipulations: raise curtain, call subroutine A which raises curtain and calls subroutine B, etc., lowering curtain before returning control to the main program which again lowers the curtain, this time to its original position. Each subroutine in this chain of calls must refrain from altering its R00. See Hanoi Tower Puzzle Generalized (Appendix A) for an interesting example.

Note: HD is not interruptible at lines 166 through 168 if the printer is present and 0 ≤ (E? mod 8) ≤ 3. See ΣC instructions for an explanation and remedy.

WARNING: HD and UD are meant to be used together in running programs. If you edit or pack program memory after using HD and before using UD, don't use UD. That could cause loss of catalog 1. Lower curtain with CU instead. UD assumes HD was executed.

LINE BY LINE ANALYSIS OF HD

Regard the initial contents of status register c as the following 14 hex digits:

$$s_1s_2\ s_30\ 01\ 69\ z_1z_2\ z_3e_1\ e_2e_3$$

TECHNICAL DETAILS

XROM: 10,20 | **HD** | SIZE: X + 6

Stack Usage:
- 0 T: temporary c
- 1 Z: T
- 2 Y: Z
- 3 X: Y
- 4 L: USED

Flag Usage: FLAG REGISTER USED BUT RESTORED
04:
05:
06:
07:
08:
09:
10:
25:

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: ALTERED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

ΣREG: PREVIOUS CURTAIN

Data Registers:
R00: CONTAINS CODE FOR REGISTER c. NO OTHER REGISTER USED.
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Global Labels Called:

Direct	Secondary
ΣC (IN LINE)	NONE

Local Labels In This Routine:
NONE

Execution Time: 1.8 seconds.

Peripherals Required: NONE

Interruptible? ONLY IF PRINTER NOT ATTACHED*
Execute Anytime? NO!

Program File: ML

Bytes In RAM: 127 WITH END

Registers To Copy: 64

Other Comments:
* If printer is attached there is a 50% chance that HD cannot be interrupted. If you have trouble, change SIZE by 4.

where: $s_1s_2s_3$ = the abs. 3 hex-digit address of the first Σ-register

$z_1z_2z_3$ = the abs. 3 hex-digit address of R00

$e_1e_2e_3$ = the abs. 3 hex-digit address of reg. containing .END.

Line 144 results in transferring the contents of register X to register L. Lines 145 and 151 ensure the retention of the contents of registers Y, Z, and T. Upon exit from HD (via ΣC), what had been y, z, and t have become x, y, and z, respectively. Lines 146-150 place into the alpha-register

10 01 69 $z_1z_2z_3e_1$ e_2e_3 2A 2A

Line 152 places

10 10 01 69 z_1z_2 z_3e_1 e_2e_3

into the register specified by register L (which via line 144 was the argument passed to HD). Line 153 sets the pointer to the statistical registers to the desired location for R00 after the curtain is raised. The processing is completed using ΣC.

Routine Listing For: HD

```
143♦LBL "HD"
144 SIGN
145 RDN
146 RCL c
147 "8"
148 X<> [
149 "├**"
150 STO \
151 RDN
152 ASTO IND L
153 ΣREG IND L

154♦LBL "ΣC"
155 CLA
156 RCL c
157 STO ]
158 STO [
159 "├A"
160 CLX
161 STO \
162 "├B"
163 STO [
164 "├CD"
165 X<> ]
166 X<> d
167 SF 08
168 X<> d
169 X<> \
170 "├E"
171 X<> d
172 SCI IND \
173 X<> d
174 STO ]
175 RDN
176 RCL c
177 ΣREG 00
178 X<> c
179 STO [
180 "├F"
181 RDN
182 RCL \
183 "├GHI"
184 STO L
185 RDN
186 RCL c
187 STO [
188 X<> ]
189 "├J"
190 STO [
191 "├K"
192 X<> L
193 STO [
194 "├L"
195 X<> \
196 X<> c
197 RDN
198 CLA
199 END
```

REFERENCES FOR HD

See *PPC CALCULATOR JOURNAL*, V7N5P45 for an introductory discussion.

CONTRIBUTORS HISTORY FOR HD

The HD / UD routines were conceived and written by Keith Jarett (4360). An improved HD (see *PPC CALCULATOR JOURNAL*, V8N2P2) was written by Clifford Stern (4516) several weeks after the ROM was assembled.

FURTHER ASSISTANCE ON HD

Call Harry Bertuccelli (3994) at (213) 846-6390.
Call Keith Jarett (4360) at (213) 374-2583.

NOTES

HN - HEX TO NNN

This routine translates a hexadecimal expression of up to 14 bytes in the alpha register into a non-normalized number (NNN) of up to 14 digits in the X register. Leading zeroes need not be entered, but can be at the user's option. The contents of X, Y, and Z are preserved in Y, Z, and T respectively. The original hexadecimal expression in alpha is destroyed, and the NNN produced is left in register M as well as in X.

HN

Example 1: Set the program pointer at byte 3 of register 0CE. For this we need to store 0000000030CE into register b.

DO:	SEE:	
ALPHA (On)		(Leading 0's need not be
30CE	30CE	entered)
ALPHA (Off)	0.0000	
XEQ HN	0.0000	
XEQ GE	0.0000	
STO b	0.0000	

The program pointer is now at byte 3 of register 0CE (decimal 206). It is "looking at" the program line which begins in the next byte, that is, byte 2 of register 0CE.

Example 2: Set the flags to provide the 41C default status, except that the display mode is to be "FIX/ENG 4", under which numbers too large or too small to be displayed in FIX 4 will appear in ENG 4 rather than SCI 4.

By reference to a flag map we determine that in 41C default status, register d contains 0000002C048000, where "2" is audio enable flag 26; "C" is decimal index flag 28 and digits grouping flag 29; "4" is the number of digits flag 37; and "8" is FIX format flag 40. To provide FIX/ENG 4, we need to set both flag 40 and flag 41 which cannot be done from the keyboard; this means the "8" must be changed to a "C".

DO:	SEE:	
ALPHA (On)	2C04C000_	(Leading 0's need not
2C04C000	0.0000	be entered)
ALPHA (Off)	0.0000	
XEQ HN	0.0000	
STO d	0.0000	

(The desired flag setting is in register d. To test:)

52 EEX 9	52 9_	(e.g., a Federal deficit of $52 billion)
Enter↑	52.0000	

The display operates as intended, showing "billions" in this case, rather than the "5.2000 10" which would otherwise have appeared and necessitated mental repositioning of the decimal point.

COMPLETE INSTRUCTIONS FOR HN

HN accepts 14 or fewer hex digits in alpha and places the corresponding code in X and M. The former contents of X, Y, and Z are saved in Y, Z, and T. Lastx is cleared. The reason leading zeroes need not be entered is that whatever is right-adjusted in registers N and M at execution will, upon completion, be right-adjusted in X. Unlike its inverse NH, which will decode anything which it finds in the X register, HN expects registers N and M of the alpha register to contain only valid alphanumeric expressions of the hex values 0 through 15. In standard hex notation these are 0 through 9 and A through F (although "Space" can be used in place of 0). For example, if a "Q" is found in the alpha register, HN will not be able to ascertain what we are trying to do since "Q" is not a standard symbol for a hex value.

However, HN will accept as input the "natural language" symbols produced by its inverse NH under that routine's flag 10 option. This may be of little utility ordinarily because one of those symbols, the semi-colon (;) equivalent to "B" for a hex value of 11, cannot be entered from the keyboard and, since we must therefore enter "B" in this instance, we might as well use the alphabetical letters consistently. But there may be times when the absence of ";" makes this feature useful when we are using HN in conjunction with NH.

MORE EXAMPLES OF HN

Example 3: Move the program pointer to the middle of a multi-byte instruction to perform a quick synthetic edit without the use of the byte jumper. Let us assume we wish to delete the unnecessary 1" from "1 E4" and that this instruction is line 03 of "QU".

DO:	SEE:	
GTO "QU"	0.0000	
GTO .003	0.0000	
PRGM (On)	03 1 E4	(to Check)
PRGM (Off)	0.0000	
RCL b	0.0000 -61	
SF 10	0.0000 -61	(See NH writeup)
XEQ NH	000000000051:?	

We have used the flag 10 option of NH for speed. We now need to set the pointer one byte further (down) in this register. Without worrying about what values the : and ? correspond to, we simply:

41:?	41:?	
ALPHA (Off)	0.0000 -61	
XEQ HN	0.0000 -61	(The new address is in X)
XEQ GE	0.0000 -61	(Return to RAM at line 00)
STO b	0.0000 -61	(The new address is in b)

We now do the edit.

PRGM (On)	00 REG 127
Enter↑	01 ENTER↑

The ENTER↑ has separated the "1" from the "E 4"; we now remove both the "1" and the "ENTER↑."

GTO.003	03 1
DEL 002	02 RCL 01
SST	03 E 4 (As desired.)

This also illustrates an important feature of GE: it causes the line number in register e to be set to 000 in RAM. When this is followed by STO b it places the pointer exactly where we want it and, when this is followed by PRGM (On), the pointer stays there. If we used CAT 1 instead of XEQ GE to get to RAM, the line number would be set to "FFF" and PRGM (On) would cause the pointer to go back to the beginning of "QU" and then count forward again to the beginning of the 1 E4 line, negating our efforts.

The quick synthetic editing capability provided by the use of NH and HN in conjunction with GE can be used for any editing job where the byte jumper or enhanced byte jumper will work, and does not require the entry of a byte jumper controller, or any key assignments other than RCL b and STO b. (Compare PA).

APPLICATION PROGRAM 1 FOR HN

Although HN was written without a prompt in order to meet the original ROM specifications, members who have been using RAM "CODE" routines may miss this convenience. The following is a suggested RAM access routine for HN, based upon a discovery of Bill Wickes's, which provides a prompt that "hangs" in the display while the first several characters are being entered.

```
01 LBL "CODE"      08 STOP
02 hex F2 04 80    09 STO d
03 RCL M           10 RDN
04 X<>d            11 XROM HN
05 "CODE="         12 XROM T1
06 AVIEW           13 END
07 CLA
```

The 41C is left in exactly the same configuration as with HN alone, i.e., exactly as its inverse NH expects to find it. As a variant of the above, replace line 05 with "bCODE= "(Note the space). The result is a prompt for "bCODE", that is, for 4 bytes designed to set the program pointer by means of STOb. So long as no more than 4 bytes are entered, the "b" remains in the display, indicating that valid register b contents are being entered. But as soon as a fifth byte is entered, the "b" disappears, signalling us that the NNN being produced is no longer suitable for storing into register b, and that we are generating a longer NNN for some more general purpose.

LINE BY LINE ANALYSIS OF HN

The flag sequence at lines 64-108 is repeated 7 times by the DSE L loop loaded by SIGN at line 63. On each iteration 2 alphanumeric bytes are "positioned at" flags 00-15 of register d, and "compressed" into digits by being shifted to the right (rather than to the left as in most earlier versions of "CODE") with the result that upon exit from register d at line 102, the leftmost byte is the one to be discarded and the second byte is the one to be kept. Lines 68 to 80 examine the original "righthand" byte at flags 08-15. Since the bits "in" flags 12-15 do not need to be shifted if the value is numeric (0-9) an immediate jump at line 71 is taken in that case. Lines 72-80 thus operate only on the A-F values, taking the jump at line 76 if it is B, D, or F. Lines 77-80 convert the A, C, or E values. Lines 81-100 examine the original "lefthand" byte; since the bits at flags 04-07 do need to be shifted to flags 08-11, this sequence is necessarily longer than the first one. Lines 82-89 do the shifting and take the jump at line 91 if the value is numeric (0-9). Lines 92-100 operate only on the A-F values, doing the same thing for this byte which lines 72-80 did for the other one. Lines 102-108 reformat the alpha register for the next iteration, while building the desired NNN in both N and X, so that at line 111 the completed NNN is in X, where we ultimately want it, and is written into the alpha register at line 111-112 only for viewing purposes.

Erroneous Entry Results

As mentioned above, HN expects the alpha register to contain either standard hex notation (0-9 and A-F) or one of the "natural language" symbols (: ;<=>?). There are 38 other characters which can be entered from the keyboard in ALPHA mode. The following table shows how HN will evaluate each of these erroneous entries:

Character Erroneously Entered for HN	Evaluated by HN as:
(Space)	Ø
$,	4
%,	5
H, P, X,	9
I, Q, Y, a, *,	A (10)
J, R, Z b, +,	B (11)
G, K, O, S, W, c, ,	C (12)
L, T, d, -, ≠, ∡ ,	D (13)
M, U, e, .,	E (14)
N, V, /, Σ, ↑,	F (15)

Note that this table differs radically from that published by Kai Albertsen (6747) in the *PPC CALCULATOR JOURNAL*, V8N5P6. This is because the flag logic sequence in HN differs from that in the version of "CODE" that Kai analyzed. Note also that the "space" can be handy one-stroke substitute for the two-stroke "0".

CONTRIBUTORS HISTORY FOR HN

The original "CODE" was conceived by William C. Wickes (3735) and appeared in December 1979 in the *PPC CALCULATOR JOURNAL*, V6N8P29. It, and the other Wickes "black box" programs, opened the door to synthetic programming by permitting the exploration of status and program registers and by demonstrating the power of synthetic instructions which the programs themselves contained. Bill published a new version of "CODE" in March 1980 in the *PPC CALCULATOR JOURNAL*, V7N2P35 and by late 1980 had successively developed improved revisions which were shared with other members. In May 1980 Valentin Albillo (4747) published in the *PPC CALCULATOR JOURNAL*, V7N4P28 a variant which he called "R". The first "CODE" routine proposed for the ROM (see *PPC CALCULATOR JOURNAL*, V7N7P16) was called "H↑N" and was the then latest of Bill Wickes's revisions. In October 1980, Synthetic Coordinator, Keith Jarett, published in the *PPC CALCULATOR JOURNAL*, V7N8P9 a version called "CO" by Phillipe Roussel(4367), but announced that the ROM would instead contain an HN by George Eldridge (5575) using the already proposed routine "D↑C". Both contributions were noteworthy because they used arithmetic manipulation rather than relying exclusively on flag manipulation within register d. Phillipe's was the shortest (17 seconds) so far developed. In December 1980, as the ROM "CODE/DECODE competition" heated up, Keith published in the *PPC CALCULATOR JOURNAL*, V7N10P16 a 10.5 second version of HN written by Steven R. Jacobs (5358). It too combined a flag register sequence with arithmetic manipulation. A somewhat similar routine by Gerard Westen (4780) was not published. The version finally selected was written by Richard H. Hall (4803) and received a final edit by Roger Hill (4940). It is

TECHNICAL DETAILS

XROM: 10,41	HN	SIZE: 000

Stack Usage:
- 0 T: Z
- 1 Z: Y
- 2 Y: X
- 3 X: CODE
- 4 L: CLEARED

Flag Usage: MANY USED BUT ALL RESTORED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10:
- 25:

Alpha Register Usage:
- 5 M: CODE
- 6 N: CLEARED
- 7 O: CLEARED
- 8 P: CLEARED

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R06:
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine:
2
14 TWICE

Execution Time: 5-8 seconds.

Peripherals Required: NONE

Interruptible? YES*

Execute Anytime? NO

Program File: NH

Bytes In RAM: 107 WITH END

Registers To Copy: 33

Other Comments:
*Interruption may halt execution in PRGM MODE. In this case, switch to RUN MODE before pressing R/S to continue.

not only the fastest "CODE" ever devised (microcode aside), but it's the only one that saves any stack registers, let alone three.

FURTHER ASSISTANCE ON HN

Call Richard H. Hall (4803) at (301) 383-1214.
Call Steven Jacobs (5358) at (801) 484-3672.

Routine Listing For: HN

```
61♦LBL "HN"
62 7
63 SIGN

64♦LBL 02
65 RDN
66 RCL \
67 X<> d
68 CF 11
69 CF 10
70 FC?C 09
71 GTO 14
72 SF 12
73 FC?C 15
74 SF 15
75 FS? 15
76 GTO 14
77 FC?C 14
78 SF 14
79 FC? 14
80 SF 13

81♦LBL 14
82 FS? 07
83 SF 11
84 FS? 06
85 SF 10
86 FS? 05
87 SF 09
88 FS? 04
89 SF 08
90 FC? 01
91 GTO 14
92 SF 08
93 FC?C 11
94 SF 11
95 FS? 11
96 GTO 14
97 FC?C 10
98 SF 10
99 FC? 10
100 SF 09

101♦LBL 14
102 X<> d
103 X<> ]
104 "⊢*"
105 STO ↑
106 "⊢*"
107 X<> ↑
108 STO ]
109 DSE L
110 GTO 02
111 CLA
112 STO [
113 AOFF
114 END
```

NOTES

APPENDIX G GLOSSARY OF TERMS

The following list of terms will aid the HP-41 user in his or her understanding of the technical aspects of the machine. Many of these terms are unknown outside of PPC, others are unofficial HP terms, and many are standard computer science or mathematics terms. The descriptions of the terms are not well researched 'definitions', but they are adequate to use this manual.

A

ABSOLUTE ADDRESS - Usually referring to a register number referenced to the "T" register, R000: "X" register, R003; e register, R015; void R016 through R191 ; Key Assignments R192; etc. to highest data register, R511. See LF Figure 1.

ALPHA STRING - A sequence of bytes preceded by the text (superscript T) prefix. The HEX (TEXT) code for the text indicator is F1 through FF. (241 thru 255 decimal). A six character alpha string has an F6 preceeding it. The TEXT 0, F0, is often used as a one byte "NOP".

ALPHA REGISTER - A group of four registers, M, N, O & P, that are "coupled" by the HP-41 to hold and "process" 28 bytes in a special manner to display ALPHA strings. The right most seven bytes is register M, the left most seven bytes is register P. Also, see M, N, O and P registers.

ASCII - American Standard Code for Information Interchange. A code for representing characters by integers. For example, the letters of the alphabet are represented by successive integers beginning with A = 65 for upper case and a = 97 for lower case. The numerals are encoded beginning with 0 = 48.

ASSEMBLY LANGUAGE - A system of writing microcode programs by means of easily remembered mnemonics. Assembly language is used by programmers because it is more readable than machine code. A large computer performs the translation and also provides some error checking.

AUSTRALIAN NOTATION - See Natural Notation.

B

BENDER - An industrial term applied to the piezoceramic transducer used to make the HP-41 TONE sound. Alarm watches use a bender to generate the alarm sound. Radio Shack sells a Mu-Ruta bender calling it a miniature piezo buzzer element as Cat. No. 273-064.

BENDER COUPLER - A flat plane device usually made of copper clad circuit board that comprises a capacitive pick-up plate. The bender coupler allows convenient non-audio pick-up of the bender signals for control, alarm, or demonstration applications.

BINARY - A system of representing numbers in base two using only the digits 0 and 1.

BIT - Binary Digit. In a computer, the data and instructions are represented by BIT patterns, i.e. 0's and 1's in memory. The 41 register is 56 BITS, which are grouped 8 at a time to make 7 bytes. Four BITS make a nybble, eight BITS make a byte.

BYTE - Eight BITS. Not all words are bytes. A 41 ROM word is 10 BITS, a RAM word is 8 BITS. The PPC ROM has 8,192 words, not 8,192 bytes. In terms of bytes, the total BITS could provide 10,240 bytes.

BYTE JUMPER - One of a class of key assignments (for example 241, 65) that advances the program pointer when executed in RUN mode. All key assignments from row F act as byte jumpers. See PPC CJ, V7N4P26 and V7N6P43.

BLOCK - A set of contiguous registers, sometimes containing a vector. A Generalized Block is a set of registers that are uniformly spaced, as a column or diagonal of a stored matrix. A block is usually defined as bbb.eeeii in the same manner as HP defines the ISG, or DSE instructions.

BOXED STAR - All 14 segments are "on" as a default display when any byte other than one of the 83 programmed "characters" are displayed. Fourteen segments could provide 16,384 "characters". The boxed star is often called a starburst by HP employees. It is typed as ⊠ in this manual.

BUGS, PPC - Unique HP-41C/CV BUGS identified and formally numbered by PPC for member identification. Over 70 BUGS are known, only 9 qualified for PPC classification for their utility. See BUG 1 through BUG 9.

BUG 1: Early HP-41C's didn't save LASTX when executing Σ+ and Σ-. A STO L should precede these instructions in BUG 1 machines. Also see PPC J, V6N5P27c.

BUG 2: Early HP-41C's had an indirect addressing bug that permitted storing and recalling from program memory. Arguments from 999 to 704 correspond to registers 25 thru 319. Data is normalized when recalled. See PPC J, V6N5P28b.

BUG 3: Early HP-41C's have BUG 3 if you see the BAT annunciator after:

49, STO 00
SF IND 00

BUG 3 allows the setting and clearing all 56 flags indirectly from R00 through R99. See PPC J, V6N5P28C.

BUG 4: Many HP-41C's have a BUG when computing the SIN of very small angles (≤ 5.729577951 x 10^{-99}). The HP-41 will give 1 for the SIN of 5.7... x 10^{-99}). See PPC J, V6N6P30b.

BUG 5: Incomplete CLP if 82143A printer is plugged in and on.

MODE	Max. Lines Cleared.
"NORM" or "TRACE"	233
"MAN"	1089
No Printer	1089

See PPC J, V6N6P30c.

BUG 6: HP-41C digit termination bug in translating HP-67/97 programs. An HP-67 sequence: EEX, CHS, 7, CHS, 5, CHS produces E-7-5- on one line in an HP-41 program. See PPC J, V6N8P23b.

APPENDIX G CONTINUED ON PAGE 227.

HP - HIGH RESOLUTION PLOT

This routine will generate plots on the 82143A printer in higher resolution along the length of the paper than one plot point per printed line (i.e., 7 plotted points per printed line) for between 1 and 4 functions simultaneously. Up to 9 functions may be plotted at the same time if symbols are used more than once each. High resolution means here that each of the 7 thermal dots of a printed line represents a separate plot point. Due to the interline spacing being equivalent to skipping exactly 4 thermal dot rows in the X direction, HP represents an eleven to one increase in resolution over the printer's PRPLOT or the PPC ROM's MP routines (see MP writeup elsewhere in this manual). Plot symbols are single dots, with the different functions identified by leaving various dots unplotted (more detail on this later). Plot points can occur in any of the 168 columns across the printed line.

It is recommended that the user become familiar with the MP routine before attempting to use HP. Many of the features and options of HP will be better understood with that background, since the operation of HP and MP are closely related.

HP has the following features:

A. Higher resolution than MP or PRPLOT
B. Variable plot width (1 to 168 columns)
C. Initial header information either printed or supressed
D. Standard Y-axis (12 double-width dashes) printed at beginning and end of each function plot, or replaced by pair of user-defined axes
E. Completely adjustable plot symbol usage and order
F. Four different overflow modes selectable for cases where values exceed Y min or Y max limits
G. X-axis (or axes) printed in any selected column

In addition, the following topics will also be discussed:

H. Prompting for user inputs to HP
I. Plots requiring multiple strips of printer paper - 'Superplotting'

Each of the above features and options will be explained and illustrated through detailed instructions that follow.

Basic Single-Function Plotting Example:

Use HP to plot a cosine function in high resolution. Use X limits of 0 and 360 degrees with 2-degree increments, and use Y limits of -1 and 1. The plot width shall be 168 columns.

The keystroke sequence to produce this plot is below:

KEYSTROKES	DISPLAY	RESULT
SIZE 031		
GTO . .	PACKING, REG ___	To free RAM
PRGM LBL ALPHA		
CO ALPHA	01 LBL^{T}CO	First line
COS	02 COS	Second line
XEQ END	.END. REG ___	Last line
PRGM	0.0000	Out of PRGM
XEQ RF		CF00-28
-1	-1	
STO 00	-1.0000	Y minimum
1	1	
STO 01	1.0000	Y maximum
168	168	
STO 02	168.00	Plot Width
0	0	
STO 08	0.0000	X minimum
360	360	
STO 09	360.0000	X maximum
2	2	
STO 10	2.0000	X increment
ALPHA CO	CO	
ASTO 15 ALPHA	CO	Function name
1	1	No. functions
XEQ HP		Plots the function

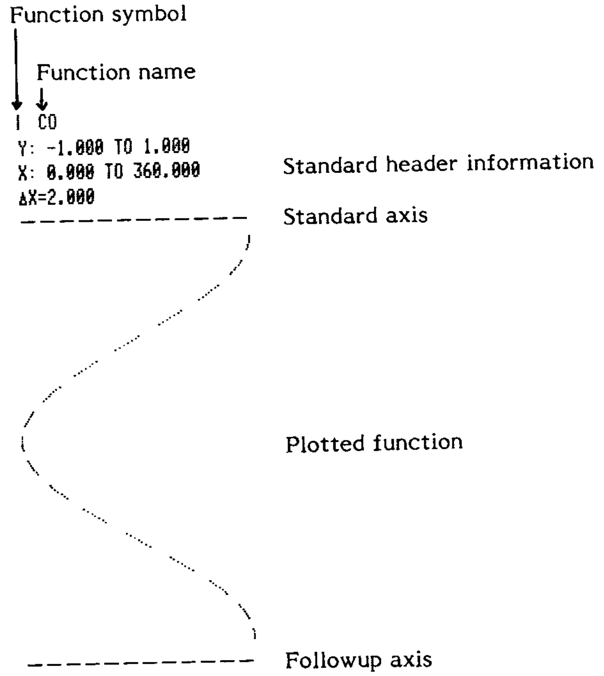


Figure 1. The cosine curve plotted using HP. Execution time: 7 min 43 sec

Basic Multiple Function Plotting Example:

Example 2. Plot the following 3 functions using HP: Y=X↑(1/2), Y=3X and Y=X↑2 over the range of X=0 to X=5. Let the X increment be X=0.03. Use Y limits of 0 and 25 and plot in all 168 columns.

Note that we will clear the user flags by calling the ROM routine RF (Reset Flags), then set flags 21 and 55 to enable the printer.

APPLICATION PROGRAM FOR:	HP
01♦LBL "SQR" 02 SQRT 03 5 04 * 05 RTN	Function #1
06♦LBL "X" 07 3 08 * 09 RTN	Function #2
10♦LBL "X-SQ" 11 X↑2 12 RTN	Function #3

13♦LBL "PLOT-3" 14 XROM "RF"	Plot routine
15 SF 21 16 55 17 XROM "IF"	Clear F00 - F28, SF21, SF55
18 0 19 STO 00	Ymin
20 25 21 STO 01	Ymax
22 168 23 STO 02	Plot width
24 0 25 STO 08	Xmin
26 5 27 STO 09	Xmax
28 .03 29 STO 10	X increment
30 "SQR" 31 ASTO 15 32 "X" 33 ASTO 16 34 "X-SQ" 35 ASTO 17	Store fcn names
36 3	No. functions
37 XROM "HP" 38 END	Call to HP

```
| SQR
⁞ X
: X-SQ
Y: 0.000 TO 25.000
X: 0.000 TO 5.000
ΔX=0.030
```

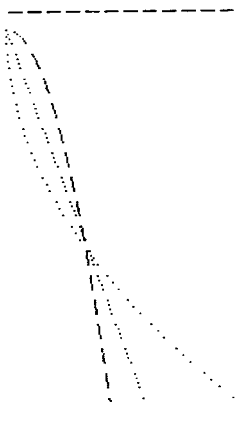


Figure 2. Simultaneous plot of the 3 functions in Example 2 using HP. Execution time: 15 min 17 sec.

A. COMPLETE INSTRUCTIONS FOR HP

Load the following registers with the required information:

R00 = Y minimum value (the Y direction is across the narrow width of the printer paper)
R01 = Y maximum value
R02 = Plot width (1 to 168 columns)
R04 = Global label (6 char's or less) of Y axis plotting routine in RAM, if used (with F09 set)
R08 = X minimum value (The X direction is down the long dimension of the printer paper).
R09 = X maximum value
R10 = X increment (delta X) value
R12 = Symbol/function map (stored only if symbol order/usage is changed and F04 set; default order is 0.123456789)
R15 = Global LBL of function #1 (6 char's or less)
R16 = Global LBL of function #2, if used
R17 = Global LBL of function #3, if used
R18 = Global LBL of function #4, if used
R19 = Global LBL of function #5
R20 = Global LBL of function #6
R21 = Global LBL of function #7
R22 = Global LBL of function #8
R23 = Global LBL of function #9

(R19–R23: Over 4 fcn's requires re-use of plotting symbols)

Rgeisters 03, 05-07, 11,13,14,24 to 39 are also used.

Registers 24 to 39 act as a 'software print buffer' to store the sorted column positions of the plot dots. These registers are filled counting from R24 up for the number of total dots used by all the symbols chosen for the functions plotted in the particular HP plot. If all 4 standard function identifiers are used, then a total of 16 registers in the software buffer will be required, using R24 to R39. If only one function is plotted using the two-dot identifier, then only R24 and R25 are required for a software buffer, and the rest are unused. If more than 16 dots are required to identify the functions plotted, the buffer extends beyond R39, up to the last required position. The limitation of the 82143A print buffer to 43 bytes restricts the maximum number of dots to be plotted to less than 21 per line, due to the bytes required in the buffer for skipping characters and columns between printed dots. More will be presented on function identifiers later.

A.1. Restrictions on Functions to be Plotted.

The functions in RAM which are plotted must accept input passed to them from HP in the X register and exit with output also in the X register. Global labels must not exceed 6 characters in length. The functions must not change the display mode without returning it to FIX 0 (the mode when the function was entered) before returning.

A.2. Flag Usage.

F10: Used internally
F09: Set if user-defined Y-axis routine is used with global label in R04;
Clear if standard Y axis (12 dashes) is desired
F08: Used internally
F07: Set if user wishes to skip standard header information (function symbols, Y limits printed);
Clear for standard header to be printed
F06 & F05 Plot overflow modes:

F06:	Clear	Set	Set	Clear
F05:	Clear	Clear	Set	Set
	Points stay at edge of the plot	Points dis-appear at edge of the plot	Points re-flected back from the edge	Points return from edge of the plot
	Clipping Mode	Disappearing Mode	Mirror Plot Mode	Wraparound Mode

F04: Set only if function identifier order/usage is changed by the user (by storing symbol map in R12;
Clear otherwise

A.3. Execution Times For HP.

Running times for the examples presented were obtained by timing the actual programs using the ROM and printer. Speed of the HP41C/CV was obtained by executing the following short routine:

LBL 01 + GTO 01

This routine was run beginning with 1 in the Y, Z and T registers and with X clear. R/S was pressed, and then pressed again after 100 seconds to establish a speed count. Results ranged from the low 1600's to middle 1700's for various 41C's, so 1700 was established as a reference count. Execution times presented for each example have been normallized to the 1700 speed count. If you have sped up your HP41, you should expect significantly faster execution times than reported below.

The following relationship was obtained for the HP routine, using nonlinear regression analysis:

Execution
time, min = -0.8905 + 0.1952*L + 0.3615*L*F

where L = Number of printed lines in the plot, and
F = Number of functions plotted simultaneously

This relationship holds for a 1700-count HP41C. Program HPT has been provided for the estimation of run times for HP plots, due to the wide range of times possible. This program will calculate estimated run times normallized to any count in the 100-second speed count test above and then executes HP. If the speed count is not known for the particular 41C being used, then simply pressing R/S at the appropriate time will assume a reference count of 1700.

Enter parameters for HP into data registers, including the number of functions in X, and then:

KEYSTROKES	DISPLAY	RESULT
XEQ HPT	COUNT?	Prompts for count
Enter count, or just press R/S for 1700 count time		Prints "EST RUN TIME:" and time, then runs HP

The listing for program HPT:

BAR CODE ON PAGE 480

APPLICATION PROGRAM FOR: HP	
01♦LBL "MPT"	
02 SF 08	
03 GTO 00	
04♦LBL "HPT"	
05 CF 08	
06♦LBL 00	Store # functions plotted in R03
07 STO 03	
08 1700	
09 "COUNT?"	
10 PROMPT	Store 1700 or count in R04
11 STO 04	
12 RCL 09	
13 RCL 08	
14 -	
15 RCL 10	Compute the number of lines to be plotted
16 /	
17 1	
18 +	
19 FS? 08	
20 GTO 01	
21 11	
22 /	
23♦LBL 01	Calculate estimated run time for HP or MP
24 RCL X	
25 RCL 03	
26 *	
27 FS? 08	
28 .09566	
29 FC? 08	
30 .3615	
31 *	
32 X<>Y	
33 FS? 08	
34 -.02144	
35 FC? 08	
36 .1952	
37 *	
38 +	
39 FS? 08	
40 .02516	
41 FC? 08	
42 -.8905	
43 +	
44 1700	
45 *	
46 RCL 04	
47 /	
48 "EST RUN TIME:"	
49 "├ "	Print run time
50 FIX 2	
51 ARCL X	
52 "├ MIN."	
53 PRA	
54 RCL 03	
55 FS? 08	
56 XROM "MP"	Call HP or MP
57 FC? 08	
58 XROM "HP"	
59 RTN	
60 .END.	

The listing for HPT appears in section A.3 of the MP routine writeup, since HPT is a subset of program MPT, which performs timing for the MP routine in a similar fashion. The barcode for HPT/MPT appears in Appendix N.

A.4. Changing Display Annunciators.

As part of the operation of HP, flag 55 (the printer existence flag) is synthetically cleared using the IF routine in order to trick the calculator into assuming that no printer is present. This speeds up non-printing operations some 20 percent, which is significant in a plot that may take several minutes to complete. During the execution of the IF routine, the display annunciators may change, such as 'RAD' coming on, or flag annunciators going on or off. This situation will remain until the HP routine stops. If the user halts execution prematurely, the annunciators will return to their original configuration. This will also reset flag 55, since the printer will now be detected to be present. Pressing R/S to restart will eventually cause HP to detect that F55 is set, and again call the IF routine to clear it, and annunciators will again change. No changes will have actually occurred to flags or to any modes.

B. Variable Plot Width.

The plot width in columns is stored by the user in register R02. This can vary from 1 to 168 columns. This feature will be used extensively in many of the examples below.

C. Skip Standard Header.

If flag 07 is clear, a standard set of initial header lines is printed before the plot. This consists of each function name and its corresponding function identifier, plus the limits in the Y and X directions along with the X increment value. Setting flag 07 causes HP to skip the header information entirely and just print the Y axis, whether it is the standard 12 dashes or a user-defined axis (to be described later). This allows another header to be substituted and printed immediately before HP is called, if the user desires.

MORE EXAMPLES OF HP

Example 3. Use HP to plot Y=1-EXP(-X) and Y=1-EXP(-2X) with the new heading: 'EQUILIBRATION CURVES:". Let X range from 0 to 5 in steps of .05. Y ranges from 0 to 1.5. Make the plot width 168 columns.

APPLICATION PROGRAM FOR:	HP
01♦LBL "eX"	Function #1
02 CHS	
03 E↑X	
04 CHS	
05 1	
06 +	
07 RTN	
08♦LBL "e2X"	Function #2
09 2	
10 *	
11 CHS	
12 E↑X	
13 CHS	
14 1	
15 +	
16 RTN	
17♦LBL "CRV"	Plot routine
18 XROM "RF"	
19 SF 21	
20 55	Clear F00 - F28,
21 XROM "IF"	SF21, SF55
22 0	
23 STO 00	Ymin
24 1.5	
25 STO 01	Ymax
26 168	
27 STO 02	Plot width
28 0	
29 STO 08	Xmin
30 5	
31 STO 09	Xmax
32 .05	
33 STO 10	X increment
34 "eX"	
35 ASTO 15	
36 "e2X"	
37 ASTO 16	Store fcn names
38 SF 07	Header skip flag
39 "EQUILIBRATION"	
40 ACA	
41 " CURVES:"	Custom header
42 ACA	
43 PRBUF	
44 2	No. functions
45 XROM "HP"	Call to HP
46 .END.	

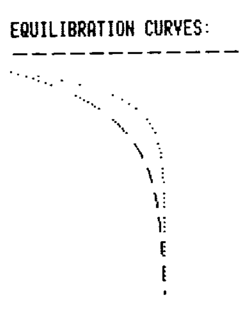


Figure 3. Two exponential decay curves plotted using HP, with a custom header replacing the standard header. Execution time: 6 min 5 sec.

FURTHER DISCUSSION OF HP

D. Custom User-Defined Y Axis and Axis Labels.

The HP routine prints a pair of standard Y axes before and after the plotted functions if flag 09 is clear. These axes consist of 12 double-wide dashes spanning the full 24 character paper width. If the user wishes his own custom Y axis, he may write an axis program in RAM, store its global label (not more than 6 characters long) in R04, and set F09 for it to be printed. This axis would print before and after the plot.

Example 4. Plot Y=-X↑2 from X=-10 to 10, with delta X=0.1. Let Y range from -100 to 0. Use a custom Y axis, labelling values at -100, -50 and 0.

APPLICATION PROGRAM FOR:	HP
01♦LBL "-X2"	Function
02 X↑2	
03 CHS	
04 RTN	
05♦LBL "PARAB"	Plot routine
06 XROM "RF"	
07 SF 21	Clear F00 - F28,
08 55	SF21, SF55
09 XROM "IF"	
10 -100	
11 STO 00	Ymin
12 0	
13 STO 01	Ymax
14 140	
15 STO 02	Plot width
16 -10	
17 STO 08	Xmin
18 CHS	
19 STO 09	Xmax
20 .2	
21 STO 10	X increment
22 SF 09	Custom axis flag
23 "AXIS"	
24 ASTO 04	Store axis name
25 "-X2"	
26 ASTO 15	Store fcn name
27 1	No. functions
28 XROM "HP"	Call to HP
29 RTN	
30♦LBL "AXIS"	Axis program
31 FS? 00	
32 GTO 00	
33 "-100 -50"	
34 "├ 0"	
35 PRBUF	Y numeric labelling
36♦LBL 00	
37 127	
38 ACCOL	
39 SF 12	
40 "-----"	
41 ACA	
42 CF 12	
43 127	
44 ACCOL	
45 SF 12	
46 ACA	
47 CF 12	
48 127	
49 ACCOL	
50 PRBUF	Axis printed
51 SF 00	
52 END	

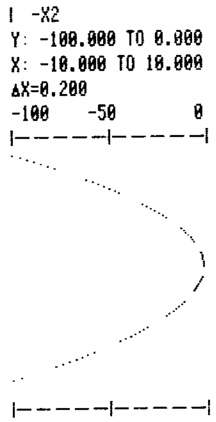


Figure 4. Plot of Y=-X↑2 using HP. Also a custom Y axis has been plotted by setting F09 and storing 'AXIS' in R04. Execution time: 4 min 0 sec.

D.1. X Value Numeric Labelling.

Just as a user-defined custom Y axis and Y-directional labelling is possible, so is numeric labelling in the X direction. This is accomplished by accumulating numeric information into the print buffer before any plot symbols are placed there. The simplest way to do this is to have function #1 of a multifunction plot place each X label into the print buffer before exiting with its function value in the X register. Of course, the plot width must decrease to allow space for these labels. If the plot width added to the label width exceeds 168 columns, HP will plot it anyway, but buffer overflow will result.

In addition, X labels must be accumulated into the print buffer carefully. Since HP executes each function 7 times before printing the buffer once, the label must only be added to the buffer once out of every 7 times the first function is executed. An incrementing counter which allows printing only when this counter reaches 7 is one way to handle this task.

D.1.1. X labelling Using ACX.

There are various ways that X labels can be placed into the print buffer. The first way is to use the printer's ACX instruction. This requires that the same number of characters are accumulated for each X label, regardless of the label's number of digits. In addition, an extra printer character position must be allotted for the sign of the X label when ACX is used, even if all the labels are positive. To assure that an equal number of printer positions is used for all labels, a test using the log of the X value may be implemented. This will yield the correct number of blank spaces which have to be skipped ahead of a label that is shorter than the maximum anticipated length. For example, if the maximum number of digits to the left of the decimal point in the labels is 4, the following log test can assure an equal number of printer positions:

X Label:	Skip amount prior to printing the label: 3-INT(LOG(ABS(X)))		
173	3-	2	=1 space
6	3-	0	=3 spaces
2204	3-	3	=0 spaces
-55	3-	1	=2 spaces

Example 5. Plot Y=sin X using HP. Use X limits of -180 and 180 degrees, with an X increment of 4 degrees. Y limits shall be -1 and 1. Label the X direction once each line with the value of the first plotted point (the topmost thermal dot).

In order to label the X direction in the example, we must allot 4 printer character positions for the labels, leaving 168 -(4*7) or 140 columns remaining for the plotted curve itself. Adding one character space between labels and graph reduces this to 133 columns. In our function to be plotted, we shall increment a numeric counter which, every 7 counts, causes the accumulating of the current X value into the print buffer. In the other 6 times through, no ACX will occur. When the current line is ready to be printed, the X label of the first plot point in the line will already have been placed in the buffer. Since the ACX must occur the first time the function is called for each printed line, we shall start the counter at its final value, and have the ISG instruction cause a branch to instructions which re-initialize the counter to count from 1 to 7.

APPLICATION PROGRAM FOR: HP	
01♦LBL "LAB" 02 XROM "RF"	Plot routine
03 SF 21 04 55 05 XROM "IF"	Clear F00 - F28, SF21, SF55
06 7.007 07 STO 36	Nth dot counter
08 -1 09 STO 00	Y minimum
10 CHS 11 STO 01	Y maximum
12 133 13 STO 02	Plot Width
14 CF 29 15 -180 16 STO 08	X minimum
17 CHS 18 STO 09	X maximum
19 4 20 STO 10	X increment
21 "SINE" 22 ASTO 15	Function name
23 1	No. functions
24 XROM "HP"	Call to HP
25 RTN	
26♦LBL "SINE"	Function
27 ISG 36 28 GTO 00	Skip labelling if not the first X value in the line
29 STO Y 30 ABS 31 X=0? 32 SIGN 33 LOG 34 INT 35 2 36 - 37 SKPCHR 38 RDN	
39 ACX	Accumulate X label
40 1.007 41 STO 36 42 RDN	Reset label counter
43 1 44 SKPCHR 45 RDN	Skip a space
46♦LBL 00 47 SIN 48 END	

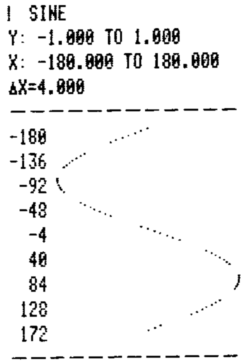


Figure 5. Plot of Y=sin X from Example 5 using HP. The X directional numeric labels correspond to the first plot point of each printed line. Note that since the X increment is 4 degrees, the numeric labels are incremented by 44 degrees due to the fact that the first plot point in each row is 11 points from the previous upper plot point. Execution time: 4 min 16 sec.

The user may desire the X labels to designate a different plot point of the 7, printed each line, such as the middle dot. The only modification to the above example necessary would be to begin the counter in R36 at a value of 4.007 rather than 7.007. This way, the 4th, or central dot in each line would have its value accumulated into the buffer. It might also be convenient to indicate to which dot the label corresponds. For the central dot, this may be done with the dash, or minus sign character, since it prints in the middle row of 7 dots. The dash would be accumulated in place of the extra blank position between the labels and the plotted function.

Example 6. Plot the Y=sin X function of Example 5, but label the X direction with the value of the middle dot in the column. Also, place a dash after the label in order to indicate that the central dot is the one labelled. Use the same X and Y limits as in the previous example.

APPLICATION PROGRAM FOR:	HP
01♦LBL "LAB"	Plot routine
02 XROM "RF"	
03 SF 21	Clear F00 - F28,
04 55	SF21, SF55
05 XROM "IF"	
06 4.007	
07 STO 36	Nth dot label counter
08 -1	
09 STO 00	Ymin
10 CHS	
11 STO 01	Ymax
12 133	
13 STO 02	Plot width
14 CF 29	
15 -180	Xmin
16 STO 08	
17 CHS	
18 STO 09	Xmax
19 4	
20 STO 10	X increment
21 "SINE"	
22 ASTO 15	Store fcn name
23 1	No. functions
24 XROM "HP"	Call to HP
25 RTN	
26♦LBL "SINE"	Function
27 ISG 36	Skip labelling if
28 GTO 00	not the middle
29 STO Y	value in the line
30 ABS	
31 X=0?	
32 SIGN	
33 LOG	
34 INT	
35 2	
36 -	
37 SKPCHR	
38 RDN	
39 ACX	Accumulate X label
40 1.007	
41 STO 36	Reset label counter
42 45	Add a dash
43 ACCHR	after label
44 RDN	
45 RDN	
46♦LBL 00	
47 SIN	Execute sine
48 END	

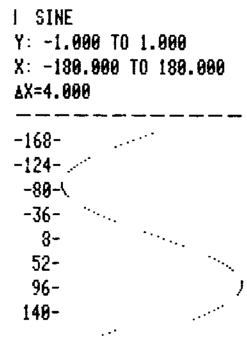


Figure 6. Plot of the Y=sin X function from X=-180 to +180 degrees using HP. The middle dot of each printed line has been identified by X numeric labels. The increment between labels remains at 44 degrees, as in Example 5, but the labels themselves are all offset by the value of 3 plot points. Execution time: 4 min 16 sec.

Note that the last printed line contains only 3 thermal dots, and thus never reaches the middle, or X labelled dot. As a result, the plotted points are not pushed to the right by the width of the X label, as in all the lines above. In order to avoid this occurrence, the user is recommended to adjust the X maximum value so that the last printed line always incorporates the X labelled value.

D.1.2. X Labelling Using CP.

Another way to label the X direction is to use the ROM routine CP to align the column of numeric X labels. See page 98 for the CP writeup in this manual. The skip index for CP, which is stored in R06, is placed there only after the original contents of R06 is saved in another register, so HP is not adversely affected. This can be done in the function which accumulates the X labels.

Example 7. Plot the 3 functions Y=X, Y=X↑2, and Y=X↑3 simultaneously using HP. Use X limits of 0 and 5 with an increment of 0.05. Y limits shall be 0 and 125. Label the first X numeric value in each printed line using CP.

APPLICATION PROGRAM FOR:	HP
01♦LBL "X"	Function #1
02 ISG 40	Skip labelling if
03 GTO 00	not the first X
04 STO 41	value in the line
05 RCL 06	
06 STO 42	Save X, Save R06

07 0	
08 STO 06	Set CP skip index
09 RCL 41	
10 FIX 2	
11 XROM "CP"	Call to CP
12 RCL 42	
13 STO 06	Restore R06
14 1.007	
15 STO 40	Reset label counter
16 RCL 41	Restore X
17 FIX 0	
18♦LBL 00	
19 RTN	
20♦LBL "X↑2"	Function #2
21 X↑2	
22 RTN	
23♦LBL "X↑3"	Function #3
24 3	
25 Y↑X	
26 RTN	
27♦LBL "3-F"	Plot routine
28 XROM "RF"	Clear F00 - F28,
29 SF 21	SF21, SF55
30 55	
31 XROM "IF"	
32 CF 29	
33 7.007	
34 STO 40	Nth dot label counter
35 0	
36 STO 00	Ymin
37 125	
38 STO 01	Ymax
39 133	
40 STO 02	Plot width
41 0	
42 STO 08	Xmin
43 5	
44 STO 09	Xmax
45 .05	
46 STO 10	X increment
47 "X"	
48 ASTO 15	
49 "X↑2"	
50 ASTO 16	
51 "X↑3"	
52 ASTO 17	Store fcn names
53 3	No. functions
54 XROM "HP"	Call to HP
55 END	

```
| X
: X↑2
: X↑3
Y: 0.000 TO 125.000
X: 0.000 TO 5.000
ΔX=0.050
```

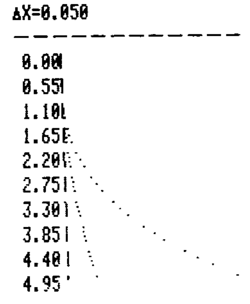


Figure 7. Plot of the 3 functions of Example 7. Since the X labels require FIX 2 display mode, a total of 5 printer character positions are needed, leaving 133 columns for the plot itself. Registers 24 through 39 are used in this example for the software print buffer, so registers R40, R41 and R42 are utilized for the X label counter, the temporary X value, and temporary R06 value respectively, during the execution of CP. Execution time: 8 min 2 sec.

D.1.3. X Labelling Using ACA.

A third approach to X-direction labelling is the use of the ACA printer function, printing X labels directly from ALPHA. The advantage of using ACA is that X labels do not require an extra blank printer position skipped for a negative sign. This obviously will allow for a wider plot field. Since HP already uses the M, N and O registers for numeric counters, however, these must be preserved during the ACA process. One solution is to store M, N and O into the stack and then return them after the X label has been accumulated.

Example 8. Plot the function Y=(X-5)↑3 from X=4 to X=7 using HP. X increment shall be 0.03. Y limits shall be -1 and 8. Label the X direction with the value of the first printed dot, and use ACA to accumulate the label into the buffer.

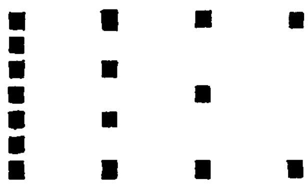
APPLICATION PROGRAM FOR:	HP
01♦LBL "X-5"	Function
02 ISG 31	Skip labelling if
03 GTO 00	not the first X
04 RCL [	value in the line
05 RCL \	Save M, N and O
06 RCL]	in Z, Y and X
07 FIX 2	
08 CLA	
09 ARCL T	
10 ACA	Accumulate X label
11 STO]	
12 RDN	
13 STO \	
14 RDN	
15 STO [	Restore M, N and O
16 RDN	
17 1	
18 SKPCHR	Skip a space
19 RDN	
20 FIX 0	
21 1.007	
22 STO 31	Reset label counter
23 RDN	
24♦LBL 00	
25 5	
26 -	
27 3	
28 Y↑X	
29 RTN	
30♦LBL "XAC"	Plot routine
31 XROM "RF"	
32 SF 21	Clear F00 - F28,
33 55	SF21, SF55
34 XROM "IF"	
35 7.007	
36 STO 31	Nth dot label counter
37 -1	
38 STO 00	Ymin
39 8	
40 STO 01	Ymax
41 133	
42 STO 02	Plot width
43 4	
44 STO 08	Xmin
45 7	
46 STO 09	Xmax
47 .03	
48 STO 10	X increment
49 "X-5"	
50 ASTO 15	Store fcn name
51 1	No. functions
52 XROM "HP"	Call to HP
53 END	

```
I X-5
Y: -1.000 TO 8.000
X: 4.000 TO 7.000
ΔX=0.030
------------
4.00 \
4.33  \
4.66  \
4.99  |
5.32  \
5.65   \
5.98    `.
6.31       `..
6.64           `....
6.97                 ..
------------
```

Figure 8. Plot of the function of Example 8. X labels (of the first printed dot in each line) have been accumulated into the print buffer using ACA, thus allowing the extra character for the X value negative sign to be eliminated. Registers M, N and O are saved in the stack while ACA is performed. Execution time: 4 min 15 sec.

E. Standard Function Identifiers and Order.

The standard set of 4 function identifiers for **HP** are shown in figure 9.



#1 2 3 4

Figure 9. The 4 standard function identifiers for **HP**. Each '■' represents a filled thermal dot. While for each function plotted, each dot represents a single plot point, different functions are identified by which dots are 'on' and which are 'off'. The function using the first identifier will have all 7 of its points plotted; the function using the second identifier will have every other point plotted, etc.

E.1. Remapping Identifier Order/ Changing Identifier Usage.

If **HP** plot function names are simply placed in the appropriate registers, the plotted functions will use the plot identifiers in the order shown in figure 9. Identifier order and usage are entirely controllable by the user however. Register R12 contains the 'symbol map' - a decimal number that matches the Nth digit to the Nth function to be plotted. If R12 isn't specified by the user, then **HP** sets it to a value of 0.123456789. Although there are only 4 standard function identifiers in **HP**, this default value for R12 is used, since most of the program lines from **HP** are shared with ROM routine **MP** as well. For the first 4 functions in **HP**, this is the standard order of identifiers. However, this can easily be changed in the RAM program which calls **HP**, or from the keyboard before pressing XEQ **HP**. For instance, if 4 functions are to be plotted using the 4 identifiers in reverse order, than set flag F04 and store 0.4321 into R12 before executing **HP**. If 6 functions all are to use identifier #3 then store .333333 in R12, SF04 and XEQ **HP**.

Since there are only 4 different identifiers for **HP**, a plot of more than 4 simultaneous functions requires re-use of the identifiers. If each of the identifiers has been used once, then 7+4+3+2, or 16 different columns may contain printed dots. This means that perhaps 32 or more print buffer positions may be required to print any single line of the plot. If identifiers have been chosen such that fewer dots are turned on than the 7 dots in identifier #1, then as many as 9 functions may be plotted simultaneously without causing premature buffer overflow.

Example 9. Plot the following 8 functions simultaneously using **HP**: Y=sin X, -sin X, cos X, and -cos X, and the same 4 as above with half the amplitude. Use function identifier #4 for all 8 functions. Use X limits of 0 and 360 degrees, with an X increment of 2 degrees. Use the full 168-column plot width, with Y limits of -1 and 1.

APPLICATION PROGRAM FOR: HP	
01♦LBL "S"	Function #1
02 SIN	
03 RTN	
04♦LBL "-S"	Function #2
05 SIN	
06 CHS	
07 RTN	
08♦LBL "CO"	Function #3
09 COS	
10 RTN	
11♦LBL "-C"	Function #4
12 COS	
13 CHS	
14 RTN	
15♦LBL "S/2"	Function #5
16 SIN	
17 2	
18 /	
19 RTN	
20♦LBL "-S/2"	Function #6
21 SIN	
22 CHS	
23 2	
24 /	
25 RTN	
26♦LBL "C/2"	Function #7
27 COS	
28 2	
29 /	
30 RTN	
31♦LBL "-C/2"	Function #8
32 COS	
33 CHS	
34 2	
35 /	
36 RTN	
37♦LBL "8PLT"	Plot routine
38 XROM "RF"	Clear F00 - F28,
39 SF 21	SF21, SF55
40 55	
41 XROM "IF"	
42 -1	
43 STO 00	Ymin
44 CHS	
45 STO 01	Ymax
46 168	
47 STO 02	Plot width
48 0	
49 STO 08	Xmin
50 360	
51 STO 09	Xmax
52 2	
53 STO 10	X increment
54 .44444444	
55 STO 12	Store symbol map
56 SF 04	
57 "S"	
58 ASTO 15	
59 "-S"	
60 ASTO 16	

61 "CO" 62 ASTO 17 63 "-C" 64 ASTO 18 65 "S/2" 66 ASTO 19 67 "-S/2" 68 ASTO 20 69 "C/2" 70 ASTO 21 71 "-C/2" 72 ASTO 22	Store fcn names
73 8	No. functions
74 XROM "HP" 75 END	Call to HP

```
: S
: -S
: CO
: -C
: S/2
: -S/2
: C/2
: -C/2
Y: -1.000 TO 1.000
X: 0.000 TO 360.000
ΔX=2.000
```

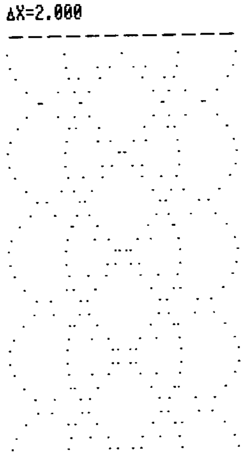


Figure 10. Plot of the 8 functions in Example 9 using HP. Since all 8 functions use identifier #4, only 16 thermal dots are required to be printed in any one line. This does not overload the print buffer, and thus the plot is possible. However, had some of the functions used identifiers requiring more thermal dots, buffer overflow would probably occur. Execution time: 27 min 51 sec.

E.2. Function Identifier Overlap Precedence.

When two functions occupy the same column, the function using the lower numbered identifier will always be plotted, concealing the higher-numbered identifier. In general, two functions will intersect at a single thermal dot position, and that dot will be turned on if the lower numbered identifier dictates it to be on. Otherwise, it will be off. For example, if identifiers #2 (dots 1,3,5,7) and #3 (dots 1,4,7) are used for two functions which intersect at the middle (fourth) position, the intersected position will be off, since identifier #2 does not turn that dot on. If the same occurred with functions using identifiers #3 and #4 (dots 1,7 on), then #3 would turn the middle dot on.

F. Overflow Modes.

In the standard 82143A printer's PRPLOT routine, if the value of the plotted function exceeds the Y minimum or maximum specified by the user, the plot symbol is drawn at the edge of the plot field. This does not tell the user anything about the function beyond the limits of the printer paper, except that the points lie beyond this edge. In HP, the user has 4 options to choose from in dealing with function overflow. These options are controlled by the 4 combined states of flags 05 and 06, as described below:

F05 CLEAR, F06 CLEAR:
Function points remain at the edge of the print field, when overflow occurs ('Clipping Mode').

F05 CLEAR, F06 SET:
Function points disappear at the edge that they exceed ('Disappearing Mode').

F05 SET, F06 SET:
Function points are reflected back from the edge they exceed ('Mirror Plotting').

F05 SET, F06 CLEAR:
Function points wrap around and return onto the plot field from the opposite edge that they exceed ('Wraparound Plotting').

The way each of the 4 overflow modes behaves when a function exceeds the user-specified limits is shown in Figure 11. The following program was used to generate the 4 plots in Figure 11. The only change made for each plot was the state of flags F05 and F06. The Y limits were purposely chosen to be too narrow to fit the full range of the function plotted.

APPLICATION PROGRAM FOR:	HP
01♦LBL "S" 02 SIN 03 RTN	Function
04♦LBL "OVER" 05 XROM "RF" 06 SF 21 07 55 08 XROM "IF" 09 -.8	Plot routine Clear F00 - F28, SF21, SF55
10 STO 00	Ymin
11 CHS 12 STO 01	Ymax
13 168 14 STO 02	Plot width
15 0 16 STO 08	Xmin
17 360 18 STO 09	Xmax
19 2 20 STO 10	X increment
21 "S" 22 ASTO 15	Store fcn names
23 CF 05 24 CF 06	Set overflow mode
25 1	No. functions
26 XROM "HP" 27 END	Call to HP

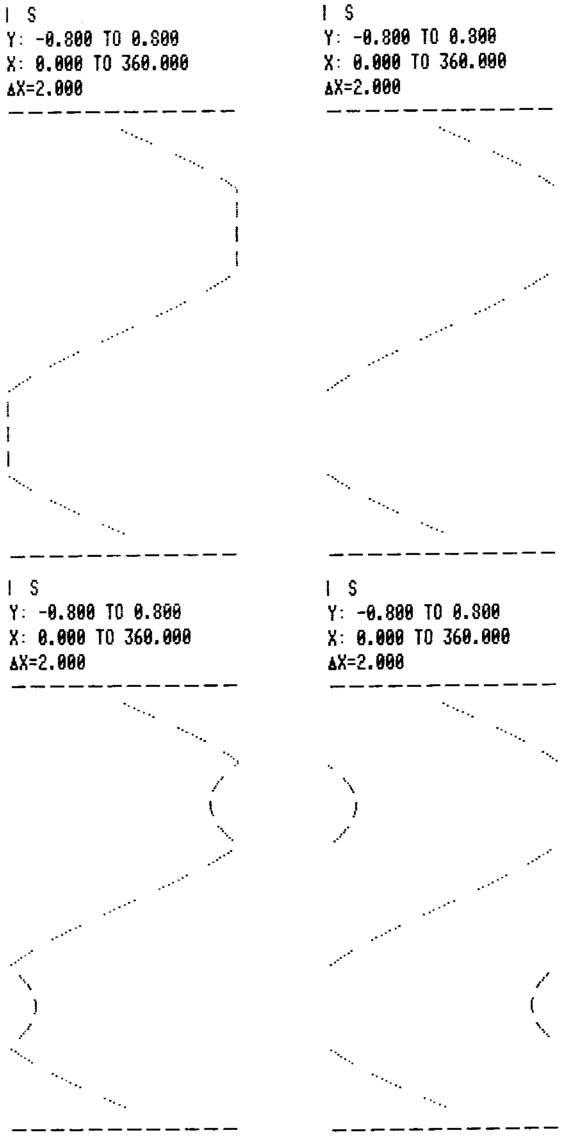


Figure 11. The four modes of overflow for HP routine shown by a sine curve drawn with identifier #1 from 0 to 360 degrees in increments of 2 degrees. Upper left: plot is clipped at the edge; upper right: plot disappears at the edge; lower left: plot reflected at the edge; lower right: plot wrapped around to the opposite edge.

F.1. Mirror Plotting.

In one of the overflow modes, plotted points are reflected back from the edge which they exceed. This was originally called mirror ploting, and was submitted as a separate routine for the PPC ROM by Frits Kuyt (236). His idea was to have a short routine that could be called by the plotted function program to reflect the overflow points. It would be compatible with all the plotting routines in the ROM, and also work with the PRPLOT printer routine. His program is listed here:

01◆LBL "MR"	
02 RCL 01	Recall Ymax
03 X<>Y	
04 X<=Y?	If value less than
05 GTO 00	Ymax, skip
06 -	
07 RCL 01	If not, reflect back
08 +	on upper edge
09◆LBL 00	
10 RCL 00	
11 X<>Y	
12 X>Y?	If value greater than
13 RTN	Ymin, RTN
14 -	
15 RCL 00	
16 +	If not, reflect back
17 END	on lower edge

Registers R00 and R01 contain the Y minimum and Y maximum values for the plot, which is true of PRPLOT, HP and MP (Multiple function low-resolution Plotting). While plotting functions using PRPLOT, if a function produces values that not only exceed a Y limit in a plot, but also exceed the opposite limit when reflected back, it would be necessary to execute the mirror plotting routine several times in succession. Otherwise, if a function plotted by PRPLOT which has already been reflected by MR exceeds the oppposite edge of the plot, it will stay at that edge.

A single call of the MR routine will actually reflect values exceeding the upper Y limit as much as a second time if they must be reflected back up from the lower edge. If a value exceeding the lower limit must be reflected a second time, however, it will stay at the edge of the plot. It is best, therefore, to call MR repeatedly to prevent this occurrence.

The version of mirror plotting built into the HP and MP routines will automatically reflect a function, no matter how many times it exceeds the plotting limits. This is illustrated in the following example.

Example 10. Plot the function Y=4X using printer program PRPLOT. Use X limits of 0 and 220 with an X increment of 11; Y limits of 0 and 200. Call the MR mirror plotting routine once in the function program. Then plot the same function using HP Use the same X and Y limits, but with an X increment of 1. Use mirror plotting mode, X labelling of the first plotted point in each line and a plot width of 133 columns.

The keystroke sequence for the PRPLOT routine is as follows:

KEYSTROKES	DISPLAY	RESULT
GTO..		
PRGM shift LBL ALPHA 4X ALPHA		
4 x XEQ MR shift RTN		
PRGM		
XEQ ALPHA PRPLOT		
ALPHA	NAME?	Prompt for function
shift 4X R/S	Y MIN?	Prompt for Ymin
0 R/S	Y MAX?	Prompt for Ymax
200 R/S	AXIS?	Prompt for axis
0 R/S	X MIN?	Prompt for Xmin
0 R/S	X MAX?	Prompt for Xmax
220 R/S	X INC?	Prompt for Xincrement
11 R/S		Prints limits, plots graph

```
01♦LBL "4X"
02 4
03 *
04 XEQ "MR"
05 END
```

```
01♦LBL "4X"
02 ISG 31
03 GTO 00
04 STO Y
05 ABS
06 X=0?
07 SIGN
08 LOG
09 INT
10 2
11 -
12 SKPCHR
13 RDN
14 ACX
15 1.007
16 STO 31
17 RDN
18 1
19 SKPCHR
20 RDN
21♦LBL 00
22 4
23 *
24 RTN
25♦LBL "P4"
26 XROM "RF"
27 SF 21
28 55
29 XROM "IF"
30 CF 29
31 0
32 STO 00
33 200
34 STO 01
35 133
36 STO 02
37 0
38 STO 08
39 220
40 STO 09
41 1
42 STO 10
43 "4X"
44 ASTO 15
45 7.007
46 STO 31
47 SF 05
48 SF 06
49 1
50 XROM "HP"
51 END
```

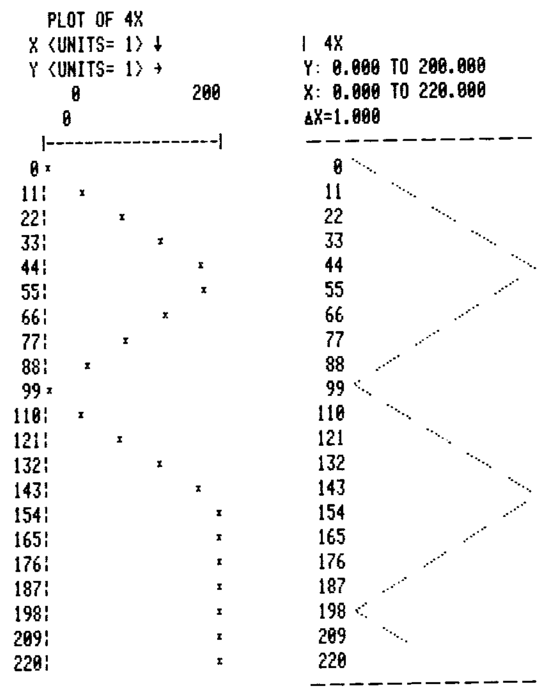


Figure 12. Two plotted versions of Y=4X, using PRPLOT in conjunction with MR mirror plotting; and using HP in mirror plotting mode. The second plot uses an X increment 1/11th of the first, so the same number of lines are plotted. Note that MR only reflects the function twice, while HP in mirror plotting mode reflects the function 4 times, and will continue to reflect if as needed. Execution times: Left: 54 sec.; right: 9 min 53 sec.

F.2. 'Disappearing' Overflow Mode.

Sometimes it its desirable for plotted function values to disappear when they exceed the Y limits of the plot. This is obtained with HP by setting flag 06 and with flag 05 clear. An example of the usefulness of this mode is in examining a specific portion of a function for its behavior, without having the nonessential sections printed.

Example 11. In order to examine their behavior at the origin, plot the following 3 functions simultaneously using HP: Y=0, Y=X↑2 and Y=2X↑3. The X limits shall be -5 and 5 with an increment value of .05. Make the Y limits -10 and +10. Use 'disappearing' overflow mode.

APPLICATION PROGRAM FOR: HP	
01♦LBL "Z" 02 0 03 RTN	Function #1
04♦LBL "X↑2" 05 X↑2 06 RTN	Function #2
07♦LBL "2X↑3" 08 3 09 Y↑X 10 2 11 * 12 RTN	Function #3
13♦LBL "P3"	Plot routine
14 XROM "RF" 15 SF 21 16 55 17 XROM "IF" 18 -10	Clear F00 - F28, SF21, SF55
19 STO 00 20 CHS	Ymin
21 STO 01 22 168	Ymax
23 STO 02 24 -5	Plot width
25 STO 08 26 CHS	Xmin
27 STO 09 28 .05	Xmax
29 STO 10 30 "Z" 31 ASTO 15 32 "X↑2"	X increment
33 ASTO 16 34 "2X↑3" 35 ASTO 17	Store fcn names
36 SF 06	Set disappearing mode
37 3	No. functions
38 XROM "HP" 39 END	Call to HP

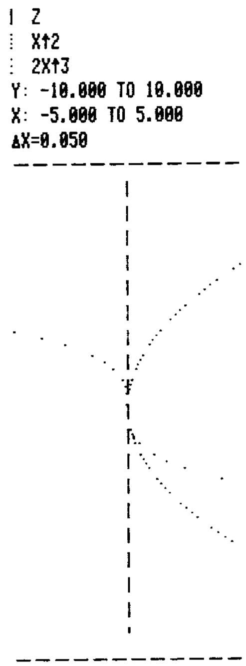


Figure 13. The 3 functions in example 11 plotted using HP in 'disappearing' overflow mode. Execution time: 18 min 17 sec.

F.3. 'Wraparound' Overflow Mode.

When the points of a function which exceed specified Y limits are of importance and are therefore to be displayed, but mirror plotting is not acceptable, 'wraparound' plotting may be used. Function points which are greater than Ymax or smaller than Ymin will be plotted coming back from the opposite edge of the plot. The wrapped around section will thus have its actual shape, but it will be shifted by a value equivalent to the width of the plot itself (Ymax - Ymin). If the function value is so large as to exceed additional limits after being wrapped around once, it will be wrapped around repeatedly, in the same fashion as mirror plotting. For this mode, flag 05 is set and flag 06 is clear.

Example 12. Plot the functions Y=2sinX and Y=5sinX from 0 to 180 degrees in increments of 1 degree . Use Y limits of -1 and 1. Use wraparound plotting mode (CF06, SF05). Use symbols 3 and 1 for the two functions respectively.

APPLICATION PROGRAM FOR:	HP
01♦LBL "2S"	Function #1
02 SIN	
03 2	
04 *	
05 RTN	
06♦LBL "5S"	Function #2
07 SIN	
08 5	
09 *	
10 RTN	
11♦LBL "PW"	Plot routine
12 XROM "RF"	Clear F00 - F28,
13 SF 21	SF21, SF55
14 55	
15 XROM "IF"	
16 SF 05	Set wraparound mode
17 -1	
18 STO 00	Ymin
19 CHS	
20 STO 01	Ymax
21 168	
22 STO 02	Plot width
23 0	
24 STO 08	Xmin
25 180	
26 STO 09	Xmax
27 1	
28 STO 10	X increment
29 "2S"	
30 ASTO 15	
31 "5S"	Store fcn names
32 ASTO 16	
33 .31	
34 STO 12	Store symbol map
35 SF 04	
36 2	No. functions
37 XROM "HP"	Call to HP
38 END	

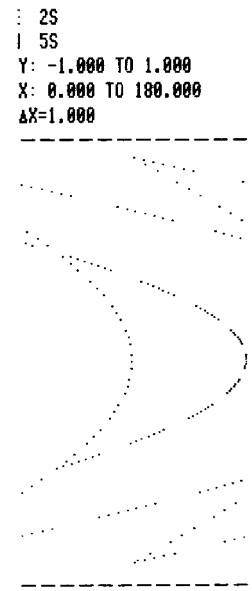


Figure 14. The two functions of Example 12 plotted by HP using wraparound plotting. Note that the functions are wrapped around repeatedly as needed, until the Y value falls within the plot field. Execution time: 12 min 44 sec.

F.4. Mixed Overflow Modes.

Another way the various overflow modes can be used is to plot different functions using different overflow conditions. The states of flags F05 and F06 can be set within the functions themselves, so that the function values plotted for each are in the necessary position according to the overflow mode prescribed. This is illustrated in Example 13.

Example 13. Plot the 3 functions of Example 11 using mirror plotting, wraparound plotting and clipped plotting in functions 1, 2 and 3 respectively, by HP. Set these overflow modes in the three function programs themselves.

APPLICATION PROGRAM FOR:	HP
01♦LBL "Z"	Function #1
02 0	
03 SF 05	Set mirror plotting
04 SF 06	mode
05 RTN	
06♦LBL "X↑2"	Function #2
07 X↑2	
08 SF 05	Set wraparound mode
09 CF 06	
10 RTN	

11♦LBL "2X↑3"	Function #3
12 3	
13 Y↑X	
14 2	
15 *	
16 CF 05	
17 CF 06	Set clipped mode
18 RTN	
19♦LBL "P3"	Plot routine
20 XROM "RF"	
21 SF 21	Clear F00 - F28,
22 55	SF21, SF55
23 XROM "IF"	
24 -10	
25 STO 00	Ymin
26 CHS	
27 STO 01	Ymax
28 168	
29 STO 02	Plot width
30 -5	
31 STO 08	Xmin
32 CHS	
33 STO 09	Xmax
34 .05	
35 STO 10	X increment
36 "Z"	
37 ASTO 15	
38 "X↑2"	
39 ASTO 16	Store fcn names
40 "2X↑3"	
41 ASTO 17	
42 3	No. functions
43 XROM "HP"	Call to HP
44 END	

```
| Z
: X↑2
: 2X↑3
Y: -10.000 TO 10.000
X: -5.000 TO 5.000
ΔX=0.050
```

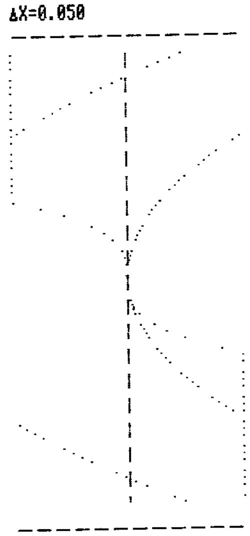


Figure 15. Plot of the 3 functions of Examples 11 and 13 using HP. Different overflow modes have been designated within the plotted functions themselves. Execution time: 17 min 57 sec.

G. X Axes in Plots.

Often when functions are plotted, it is convenient to have one or more 'axes' running along the length of the plot, at various Y heights. An example would be to have an X axis at Y=0 run down the center of a conventional sinusoidal function which ranges from Y=-1 to Y=1. One way to accomplish this is to plot a RAM function with a global label which merely returns the constant zero. However, this is unnecessary for plotting constants. All that is needed is to store the constant to be plotted into the appropriate function name register, and HP. takes care of the rest. If the number 4 is stored in R15, it will be plotted as Y=4 using the function identifier designated from R12. HP checks the contents of R15 through R23 and if a constant is present, it is plotted as an axis; if ALPHA information is present, HP searches for the corresponding global label and executes it.

Example 14. Plot the Y=sinX curve from 0 to 360 degrees with 2 degree increments. Use function identifier #2. Also plot axes at Y=1, 0, and -1 using identifiers #3, 1 and 3 respectively. Use Y limits of -1.2 and +1.2, with a plot width of 168 columns.

APPLICATION PROGRAM FOR:	HP
01♦LBL "SINE"	Function
02 SIN	
03 RTN	
04♦LBL "PRS"	Plot routine
05 XROM "RF"	Clear F00 - F28,
06 SF 21	SF21, SF55
07 55	
08 XROM "IF"	
09 -1.2	
10 STO 00	Ymin
11 CHS	
12 STO 01	Ymax
13 168	
14 STO 02	Plot width
15 0	
16 STO 08	Xmin
17 360	
18 STO 09	Xmax
19 2	
20 STO 10	X increment
21 "SINE"	
22 ASTO 15	Store fcn name
23 -1	
24 STO 16	
25 0	
26 STO 17	
27 1	
28 STO 18	
29 .2313	
30 STO 12	Store symbol map
31 SF 04	
32 4	No. functions
33 XROM "HP"	Call to HP
34 END	

```
: SINE
: -1.00
| 0.00
: 1.00
Y: -1.200 TO 1.200
X: 0.000 TO 360.000
ΔX=2.000
```

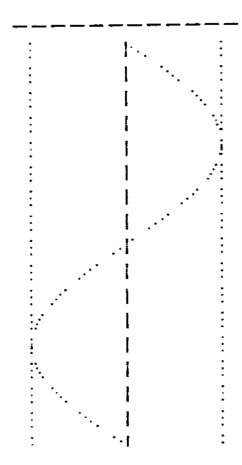


Figure 16.

Figure 16. Plot of the Y=sinX function of Example 14 using HP. Three axes have also been plotted by storing the constants 1, 0 and -1 into the function name registers R16 to R18. Execution time: 17 min 55 sec.

H. Prompting for User Inputs to HP.

Because of the large number of inputs to the HP routine, it may be inconvenient to remember where all the input information belongs. The following program provides some assistance by prompting the user for all the basic inputs to HP: function names, Ymin, Ymax, plot width, Xmin, Xmax, and X increment. It then calls the HP routine. Simply set all the flags to their correct status, set the other options appropriately and XEQ HPP. The listing is presented below:

BAR CODE ON PAGE 479

APPLICATION PROGRAM FOR:	HP
01♦LBL "MPP"	
02 SF 08	
03 GTO 00	
04♦LBL "HPP"	
05 CF 08	
06♦LBL 00	
07 "NO. FCNS?"	Input # of functions
08 PROMPT	
09 STO 04	
10 1 E3	
11 /	
12 15.014	
13 +	
14 STO 03	
15 FIX 0	
16♦LBL 01	
17 "NAME "	Input each name or X axis value
18 RCL 03	
19 14	
20 -	
21 ARCL X	
22 "⊦?"	
23 AON	
24 PROMPT	
25 FS? 48	
26 ASTO IND 03	
27 FC? 48	
28 STO IND 03	
29 ISG 03	
30 GTO 01	
31 "Y MIN?"	Input Ymin
32 PROMPT	
33 STO 00	
34 "Y MAX?"	Input Ymax
35 PROMPT	
36 STO 01	
37 "PLOT WIDTH?"	Input plot width
38 PROMPT	
39 STO 02	
40 "X MIN?"	Input Xmin
41 PROMPT	
42 STO 08	
43 "X MAX?"	Input Xmax
44 PROMPT	
45 STO 09	
46 "X INC?"	Input X increment
47 PROMPT	
48 STO 10	
49 RCL 04	
50 FIX 4	
51 FS? 08	
52 XEQ "MP"	Calls MP or HP
53 FC? 08	
54 XEQ "HPT"	
55 RTN	
56 .END.	

This routine may also be used for passing input to MP by pressing XEQ MPP. In that case, MP would be executed as the final step. If estimated execution times are also desired, one could replace the lines XROM HP and XROM MP with XEQ HPT and XEQ MPT respectively. Then, after all prompting, the run time would be printed before the plot routine was executed.

The barcode for HPP/MPP appears in Appendix N.

I. Plots using Multiple Paper Widths - 'Superplotting'.

When higher plot resolution is desired in the Y direction (across the printer paper) than can be obtained with 168 columns, it is possible to plot graphs with HP which require multiple widths of printer paper. This has been referred to as 'superplotting'. The routine shown below takes care of the housekeeping involved in printing each section of the plot, re-initializes the inputs and increments the Y limits. The only difference between the inputs for this program and for HP is that Ymax is stored in R42 instead of R01, and a Y increment value (the desired width of each printed plot section) is stored in R43. After all the function names are stored, simply set the limits and XEQ SHP:

1. Place the function names (and axis values) in R15 and up
2. Set disappearing overflow mode (CF05, SF06) so functions jump from strip to strip
3. Store Xmin, Xmax and Xinc in R08, R09 and R10
4. Store plot width in R02
5. Store Ymin in R00, Ymax in R42 and Yinc in R43
6. Enter the number of functions to be plotted
7. XEQSHP, and the plot is printed, a strip at a time, moving from Ymin to Ymax, in steps equal to the Y increment stored in R43.

The SHP listing is as follows:

BAR CODE ON PAGE 479

APPLICATION PROGRAM FOR:	HP
01♦LBL "SMP"	MP superplotting
02 STO 38	Save # fcns in R38
03 RCL 08	
04 STO 37	Ymin in R37
05 RCL 00	
06 RCL 36	
07 +	
08 STO 01	Ymin + Y increment
09♦LBL 00	
10 RCL 38	Restore # fcns
11 XEQ "MP"	Call to MP
12 RCL 01	
13 RCL 35	
14 X<=Y?	
15 RTN	If done, stop
16 RDN	
17 STO 00	
18 RCL 36	If not, increment Ymin, Ymax
19 ST+ 01	
20 RCL 37	
21 STO 08	
22 GTO 00	
23♦LBL "SHP"	HP superplotting
24 STO 45	Save # fcns in R45
25 RCL 08	
26 STO 44	X min in R44
27 RCL 00	
28 RCL 43	
29 +	
30 STO 01	Ymin + Y increment
31♦LBL 01	
32 RCL 45	Restore # fcns
33 XEQ "HP"	Call to HP

34 RCL 01 35 RCL 42 36 X<=Y? 37 RTN	If done, stop
38 RDN 39 STO 00 40 RCL 43 41 ST+ 01 42 RCL 44 43 STO 08 44 GTO 01 45 END	If not, increment Ymin, Ymax

Note that the SHP program listing also includes SMP, which is the superplotting routine for MP. See the MP writeup elsewhere in this manual. The barcode for SHP/SMP appears in Appendix N.

The first plot strip has Ymin = Ymin and Ymax = Ymin + Yinc. The next strip has Ymin = the previous Ymax and Ymax = (new Ymin) + Yinc. This process repeats until the current Ymax exceeds that which was stored in R42. If Yinc is not chosen properly, the last plot strip will exceed the designated upper limit in the Y direction, but the excess may be removed by the user with a scissors if so desired.

Example 15. Use HP superplotting to plot the following 2 functions: Y=X↑4 - 20*X↑2 + 64 and Y = X↑3 - 9X simultaneously. Use Y limits of -100 and +100 with a Y increment of 66.67 (3 strips wide). Let the X limits be -5 and +5 with an X increment of 0.02. Use function identifiers #1 and #2 for the 2 functions and also plot X axes at Y=-3, Y=0 and Y=+3 using identifier #3 for each.

APPLICATION PROGRAM FOR:	HP
01♦LBL "X4" 02 STO Y 03 4 04 Y↑X 05 X<>Y 06 X↑2 07 20 08 * 09 - 10 64 11 + 12 RTN	Function #1
13♦LBL "X3" 14 STO Y 15 3 16 Y↑X 17 X<>Y 18 9 19 * 20 - 21 RTN	Function #2
22♦LBL "SP2"	Plot routine
23 XROM "RF" 24 SF 21 25 55 26 XROM "IF"	Clear F00 - F28, SF21, SF55
27 SF 06	Set disappearing mode
28 "X4" 29 ASTO 15 30 "X3" 31 ASTO 16	Store fcn names
32 -50 33 STO 17 34 0 35 STO 18 36 50 37 STO 19 38 .12444 39 STO 12	Store symbol map

40 SF 04 41 -5 42 STO 08	Xmin
43 CHS 44 STO 09	Xmax
45 .02 46 STO 10	X increment
47 168 48 STO 02	Plot width
49 -100 50 STO 00	Ymin
51 CHS 52 STO 42	Ymax
53 66.67 54 STO 43	Y increment
55 5	No. functions
56 XEQ "SHP" 57 END	Call to SHP

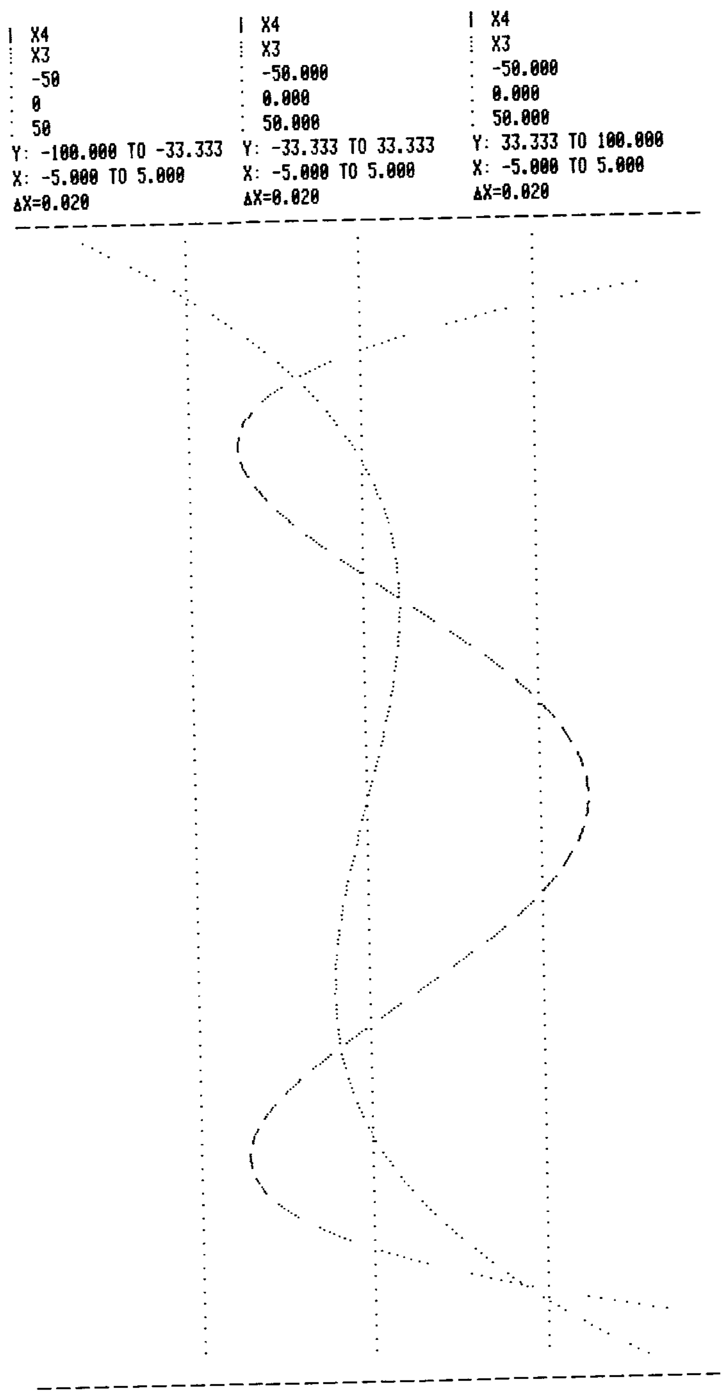


Figure 17. Plot of the 2 functions in Example 15 using SHP superplotting. The 3 strips were cut by hand and attached together edge to edge. All three strips used the full 168 column width. Execution time: 3 hr 13 min 37 sec.

Example 16. Use HP superplotting to plot all 6 trig functions (sine, cosine, tangent, cotangent, secant, cosecant) simultaneously. Use X limits of 0 and 360 degrees with an X increment of 0.5 degrees. Use Y limits of -5 and +5 with a Y increment of +2 (5 strips wide). Also plot an axis at Y=0. Use function identifiers #1,1,3,3,2,2,and 4 for the 6 functions and axis respectively. Load a roll of paper into the printer, start the program and then come back tomorrow....

APPLICATION PROGRAM FOR: HP	
01♦LBL "SN"	Function #1
02 SIN	
03 RTN	
04♦LBL "CS"	Function #2
05 COS	
06 RTN	
07♦LBL "TA"	Function #3
08 TAN	
09 RTN	
10♦LBL "CT"	Function #4
11 TAN	
12 X=0?	
13 1 E-5	
14 1/X	
15 RTN	
16♦LBL "SEC"	Function #5
17 COS	
18 X=0?	
19 1 E-5	
20 1/X	
21 RTN	
22♦LBL "CSC"	Function #6
23 SIN	
24 X=0?	
25 1 E-5	
26 1/X	
27 END	

01♦LBL "P6"	Plot routine
02 XROM "RF"	Clear F00 - F28,
03 SF 21	SF21, SF55
04 55	
05 XROM "IF"	
06 SF 06	Set disappearing mode
07 "SN"	
08 ASTO 15	
09 "CS"	
10 ASTO 16	
11 "TA"	
12 ASTO 17	Store fcn names
13 "CT"	
14 ASTO 18	
15 "SEC"	
16 ASTO 19	
17 "CSC"	
18 ASTO 20	
19 .1133224	
20 STO 12	Store symbol map
21 SF 04	
22 0	
23 STO 08	Xmin
24 360	
25 STO 09	Xmax
26 .5	
27 STO 10	X increment
28 168	
29 STO 02	Plot width
30 -5	
31 STO 00	Ymin
32 CHS	
33 STO 42	Ymax
34 2	
35 STO 43	Y increment
36 6	No. functions
37 XEQ "SHP"	Call to SHP
38 END	

Figure 18 (next page). Plot of all 6 trig functions simultaneously using HP. superplotting routine SHP. The five strips were attached together the next day. Execution time: over 15 hours.

LINE BY LINE ANALYSIS OF HP

Lines 01 to 05: Set status for F08 whether HP or MP has been executed.

Lines 06 to 13: Sets counter for number of functions.

Lines 14 to 17: Sets R12 for the symbol map.

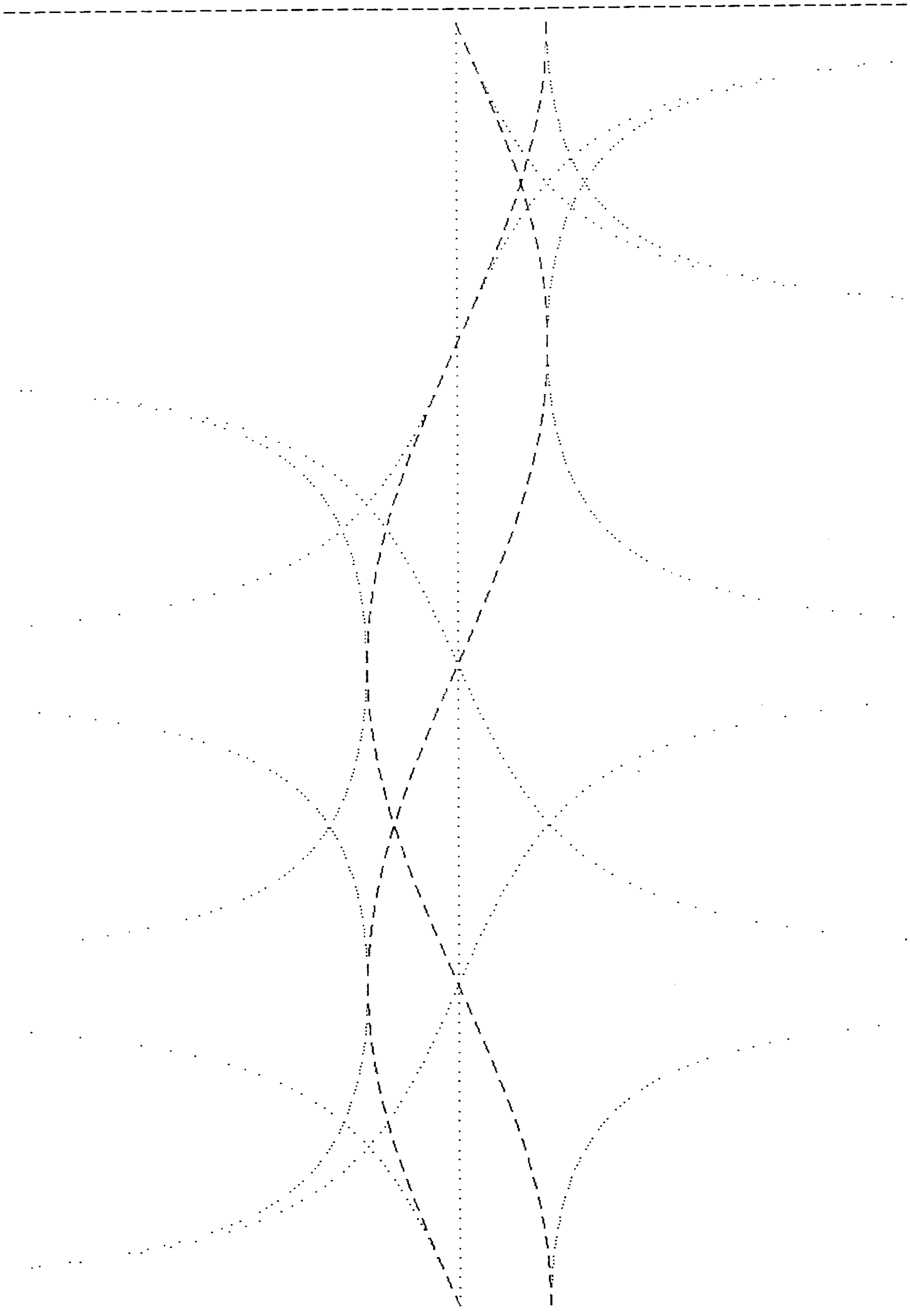
Lines 18 & 19: Related to MP value-plotting.

Lines 20 & 21: Skip header if F07 set.

Lines 22 to 40: Print symbols and function identifiers.

Continued on page

```
| SN                    | SN                    | SN                    | SN                    | SN
| CS                    | CS                    | CS                    | CS                    | CS
: TA                    : TA                    : TA                    : TA                    : TA
: CT                    : CT                    : CT                    : CT                    : CT
: SEC                   : SEC                   : SEC                   : SEC                   : SEC
: CSC                   : CSC                   : CSC                   : CSC                   : CSC
: 0.000000              : 0.                    : 0.000                 : 0.000                 : 0.000
Y: -5.000 TO -3.000     Y: -3.000 TO -1.000     Y: -1.000 TO 1.000      Y: 1.000 TO 3.000       Y: 3.000 TO 5.000
X: 0.000 TO 360.000     X: 0.000 TO 360.000     X: 0.000 TO 360.000     X: 0.000 TO 360.000     X: 0.000 TO 360.000
ΔX=0.500                ΔX=0.500                ΔX=0.500                ΔX=0.500                ΔX=0.500
```



Routine Listing For: **HP**

```
01♦LBL "MP"
02 SF 08
03 GTO 00
04♦LBL "HP"
05 CF 08
06♦LBL 00
07 14
08 +
09 E3
10 /
11 15
12 +
13 STO 11
14 RCL 12
15 FC? 04
16 .123456789
17 STO 12
18 FS? 10
19 GTO 00
20 FS? 07
21 GTO 13
22♦LBL 01
23 FRC
24 E1
25 *
26 14
27 +
28 CLA
29 XEQ IND X
30 FRC
31 E3
32 *
33 ACCOL
34 "  "
35 ARCL IND 11
36 ACA
37 PRBUF
38 RDN
39 ISG 11
40 GTO 01
41 FIX 3
42 "Y: "
43 ARCL 00
44 "⊢ TO "
45 ARCL 01
46 PRA
47 "X: "
48 ARCL 08
49 "⊢ TO "
50 ARCL 09
51 PRA
52 "ΔX="
53 ARCL 10
54 PRA
55♦LBL 13
56 XEQ 11
57♦LBL 00
58 RCL 02
59 E
60 -
61 RCL 01
62 RCL 00
63 -
64 /
65 STO 03
66♦LBL 12
67 RCL d
68 55
69 XROM "IF"
70 STO ]
71♦LBL 24
72 RCL 11
73 FRC
74 15
75 +
76 STO 11
77 RCL 12
78 STO [
79 24
80 STO \
81 FIX 0
82♦LBL 02
83 RCL [
84 FRC
85 E1
86 *
87 14
88 +
89 STO [
90 2
91 XEQ IND [
92 FS? 08
93 GTO 00
94 INT
95 RCL X
96 E5
97 /
98 1.007
99 +
100 STO 05
101 RDN
102 Y↑X
103 STO 13
104 LASTX
105 RCL 10
106 *
107 STO 07
108 E-3
109♦LBL 00
110 FRC
111 STO 06
112 RCL 08
113 STO 14
114♦LBL 03
115 RCL 14
116 RCL IND 11
117 SIGN
118 X=0?
119 GTO 00
120 LASTX
121 GTO 09
122♦LBL 00
123 RDN
124 XEQ IND L
125♦LBL 09
126 RCL 00
127 -
128 RCL 03
129 *
130 E
131 +
132 RND
133♦LBL 25
134 RCL 02
135 X<>Y
136 X<=Y?
137 X<=0?
138 GTO 00
139 GTO 13
140♦LBL 00
141 FS? 05
142 GTO 00
143 X>Y?
144 X<>Y
145 X<=0?
146 E
147 FS? 06
148 CHS
149 GTO 13
150♦LBL 00
151 FC? 06
152 GTO 09
153 X<=Y?
154 GTO 00
155 -
156 RCL 02
157 +
158♦LBL 00
159 X<0?
160 CHS
161 X=0?
162 E
163 GTO 25
164♦LBL 09
165 X<>Y
166 MOD
167♦LBL 13
168 RCL 06
169 +
170 STO IND \
171 E
172 ST+ \
173 FS? 08
174 GTO 04
175 RCL 13
176 ST* 06
177 RCL 07
178 ST+ 14
179 RCL 14
180 RCL 09
181 X<Y?
182 GTO 04
183 ISG 05
184 GTO 03
185♦LBL 04
186 ISG 11
187 GTO 02
188 E
189 ST- \
190 RCL \
191 E3
192 /
193 24
194 +
195 STO 14
196 STO 13
197 ENTER↑
198 ISG Y
199 GTO 05
200 GTO 08
201♦LBL 05
202 CLX
203 -.977
204 RCL IND Y
205 RCL Z
206 INT
207 ST+ Z
208♦LBL 06
209 RDN
210 RCL IND Y
211 X<=Y?
212 GTO 00
213 ISG Z
214♦LBL 14
215 STO IND Z
216 DSE Z
217 DSE Z
218 GTO 06
219♦LBL 00
220 ISG Z
221♦LBL 14
222 RDN
223 STO IND Y
224 RDN
225 ISG Y
226 GTO 05
227 CLX
228 STO 06
229 RCL IND 14
230 ISG 14
231♦LBL 07
232 ENTER↑
233 INT
234 RCL IND 14
235 INT
236 X=Y?
237 GTO 00
238 X<>Y
239 XEQ 10
240 RCL 14
241 STO 13
242 RCL IND X
243 GTO 08
244♦LBL 00
245 FC? 08
246 GTO 00
247 RCL 14
248 STO 13
249 R↑
250 GTO 08
251♦LBL 00
252 RCL Z
253 LASTX
254 X=Y?
255 GTO 08
256 FRC
257 ST+ IND 13
258 LASTX
259♦LBL 08
260 ISG 14
261 GTO 07
262 RCL IND 13
263 XEQ 10
264 RCL ]
265 X<> d
266 STO ]
267 PRBUF
268 FS? 10
269 RTN
270 11
271 FS? 08
272 ST/ X
273 RCL 10
274 *
275 ST+ 08
276 RCL 09
277 RCL 08
278 X>Y?
279 GTO 11
280 RCL ]
281 X<> d
282 STO ]
283 FS? 55
284 GTO 12
285 GTO 24
286♦LBL 11
287 FS? 09
288 XEQ IND 04
289 FS? 09
290 RTN
291 SF 12
292 "-------"
293 ACA
294 ACA
295 PRBUF
296 CF 12
297 RTN
298♦LBL 10
299 X<=0?
300 RTN
301 ENTER↑
302 X<> 06
303 -
304 E
305 -
306 SKPCOL
307 RCL IND 13
308 FRC
309 E3
310 *
311 ACCOL
312 RTN
313♦LBL 15
314 1.127
315 RTN
316♦LBL 16
317 2.085
318 RTN
319♦LBL 17
320 FS? 08
321 .062
322 FC? 08
323 3.073
324 RTN
325♦LBL 18
326 FS? 08
327 .042
328 FC? 08
329 6.065
330 RTN
331♦LBL 19
332 .028
333 RTN
334♦LBL 20
335 .02
336 RTN
337♦LBL 21
338 8 E-3
339 RTN
340♦LBL 22
341 RCL 33
342 RTN
343♦LBL 23
344 RCL 34
345 END
```

Note: this is also the listing for **MP**, since they share the same program steps. The Line by Line Analysis is also common to both **MP** and **HP**.

Lines 41 to 54: Print X, Y limits of plot.

Lines 55 & 56: Call the axis plot routine.

Lines 57 to 65: Set Y scaling value in R03.

Lines 66 to 70: Clear F55 and store old flag register (d) in register O.

Lines 71 to 81: Store counters in M and N registers.

Lines 82 to 132: Execute each function for the current X value.

Lines 133 to 163: Change the X values for different overflow modes.

Lines 164 to 200: Place the Y values in registers 24 and up along with the symbol ACCOL numbers.

Lines 201 to 230: Sort values in increasing order.

Lines 231 to 269: Scaling and printing line.

Lines 270 to 285: Housekeeping after a row has been printed.

Lines 286 to 297: Axis plotting routine.

Lines 298 to 312: Calculates column for given Y value.

Lines 313 to 345: ACCOL values for function identifiers (HP) and symbols (HP).

REFERENCES FOR HP

See PPC Calculator Journal, V7N1P25, V7N2P49, V7N9P17 and V7N10P11.

CONTRIBUTORS HISTORY FOR HP

The first high-resolution plot routine was presented by Jake Schwartz (1820) at the PPC West Coast Conference held in Santa Clara, California in September, 1979, and later appeared in PPCCJ in January of 1980. Nathan Meyers (4795) later improved this single-function plotting routine. Another version, by Hans-Gunter Lutke Uphues (5286), included his excellent discovery that when the printer existence flag (F55) was synthetically cleared, the speed of non-printing functions increased significantly. The final ROM version was primarily written by Tim Fischer (5793), who was instrumental in merging the HP and MP routines into a single program, in addition to adding multiple-function plotting capability to HP. Almost 20 members had some influence on either the HP program or documentation. The full list appears in the Contributors History for MP. Special thanks to Philadelphia chapter members Charles Allen (4691) and Jack Sutton (5622) for their assistance in testing and documentation for the HP routine.

TECHNICAL DETAILS

XROM: 20,29	HP	SIZE: 040

Stack Usage: 0 T: 1 Z: ALL USED 2 Y: 3 X: 4 L:	**Flag Usage:** 04: Change symbol usage 05: Plot overflow mode 06: Plot overflow mode 07: Skip standard header 08: Used internally 09: Print ustom Y axis 10: Used internally 25:
Alpha Register Usage: 5 M: 6 N: ALL USED 7 O: 8 P:	
Other Status Registers: 9 Q: NOT USED 10 ⊢: NOT USED 11 a: NOT USED 12 b: NOT USED 13 c: NOT USED 14 d: USED 15 e: NOT USED	**Display Mode:** ANY **Angular Mode:** NOT USED **Unused Subroutine Levels:** 3
ΣREG: NOT USED **Data Registers:** R00: USED R01 to R05: USED R06: USED R07: USED R08: USED R09: USED R10: USED R11: USED R12: USED R13 to R39: USED	**Global Labels Called:** Direct / Secondary IF / User Y-axis routine, User functions in RAM **Local Labels In This Routine:** 00 (10 times), 01-12, 13 and 14 (twice each), 15-19, 24 25
Execution Time: See section A.2 Execution Times for HP	
Peripherals Required: 82143A Printer	
Interruptible? YES, but slows it down slightly Execute Anytime? NO Program File: MP Bytes In RAM: 596 Registers To Copy: 86	**Other Comments:** Routine clears flag 55 by executing IF in order to speed up execution. Flag register d, with F55 set, is stored.

FINAL REMARKS FOR HP

One serious limitation to the use of HP is the small size of the HP82143A print buffer. Due to this limitation, we are not free to choose any function identifier for any or all functions plotted. Hopefully, future HP41C printers will incroporate larger buffers so any choice of identifiers will be possible for all functions simultaneously plotted.

FURTHER ASSISTANCE ON HP

Contact Jake Schwartz (1820) at 7700 Fairfield St., Phila., Penna 19152 (home phone 215-331-5324) or Tim Fischer (5793) at 7475 Morgan Rd., Bldg 11 Apt 13, Liverpool, N.Y. 13088 (home phone 315-347-6079).

NOTES

HS - HIGH RESOLUTION HISTOGRAM

This routine generates in the 82143A print buffer a high resolution bar-chart bar whose height extends from the first empty buffer column to the input value, which is scaled between 0 and 1 inclusive. The buffer is not automatically printed, so more information may be added later. The user specifies the fill character from the standard character set, to fill in 7-column sections of the bar. A list of the standard characters , along with their corresponding character numbers appears immediately after this writeup, on page 215. The remaining unfilled columns of the bar are filled by a user-defined fill-column, using the printer function ACCOL values 0 (no dots in the column) through 127 (all 7 dots filled).

Example 1. Produce a bar chart corresponding to a cosine curve through one full period, using function **HS** . Label the Y direction at -1, 0 and 1. Label the X direction at each line printed. Use X increment of 20 degrees per line. Use printer symbol number 62 (>) for the fill character and ACCOL number 8 (middle dot turned on) as a fill column.

The program for this is below, with the resulting bar chart in figure 1.

APPLICATION PROGRAM FOR:	**HS**
01♦LBL "CCC"	
02 CF 29	No commas
03 DEG	
04 .3602	
05 STO 00	Numeric counter
06 62	
07 STO 03	Fill character
08 8	
09 STO 05	Fill column
10 140	
11 STO 04	Plot width
12 2	
13 STO 06	Skip index for **CP**
14 4	
15 SKPCHR	
16 "-1"	
17 ACA	
18 53	
19 SKPCOL	Y direction
20 "0"	
21 ACA	Numeric labelling
22 58	
23 SKPCOL	
24 "1"	
25 ACA	
26 PRBUF	
27 FIX 0	
28♦LBL 00	
29 RCL 00	X label
30 XROM "CP"	Print using **CP**
31 COS	
32 1	
33 +	Cosine and scaling
34 2	
35 /	
36 XROM "HS"	Call to **HS**
37 PRBUF	Print the buffer
38 CLA	
39 ISG 00	
40 GTO 00	Return for next row
41 END	

```
   -1       0        1
  0>>>>>>>>>>>>>>>>>>>>>-
 20>>>>>>>>>>>>>>>>>>>>>>
 40>>>>>>>>>>>>>>>>>>>-
 60>>>>>>>>>>>>>>>>-
 80>>>>>>>>>>>>-
100>>>>>>>>>
120>>>>-
140>>>
160-
180
200-
220>>>
240>>>>-
260>>>>>>>>>
280>>>>>>>>>>>>-
300>>>>>>>>>>>>>>>>-
320>>>>>>>>>>>>>>>>>>>-
340>>>>>>>>>>>>>>>>>>>>>>
360>>>>>>>>>>>>>>>>>>>>>-
```

Figure 1. Bar chart of a cosine curve through a full period, using the **HS** routine.

Example 2. Plot a bar chart of both the number of attendees at each of the Philadelphia PPC chapter meetings, and the size of the mailing list at each meeting. Use different symbols to represent the two pieces of information in the same bar chart. The information for the bar chart appears in table 2 below:

Meeting No.	Mailing List	No. Attendeees
1	32	18
2	30	27
3	30	22
4	49	24
5	55	23
6	57	26
7	58	23
8	58	23
9	62	21
10	64	14
11	58	23
12	57	21
13	60	29
14	60	18
15	65	19
16	60	24

Table 2. Philadelphia PPC chapter mailing list count and meeting attendence for each of its 16 regular meetings since July, 1979.

Let us choose our 'double bars' to use printer symbol number 77 ('M') for a fill character and ACCOL number 127 (all dots filled) for a fill column for the lower part which will describe the Meeting attendence. For the upper part of the bar we can choose printer symbol number 61 ('=') for a fill character and ACCOL number 28 (middle three dots turned on) for a fill column. The Y range should be from 0 to 75. Label the X direction with the meeting number, using the **CP** routine. Label the Y direction with tic marks at 0, 10, 20, 30, ..., 70, 75.

The number of characters needed for labeling the X direction will be 3 for the meeting number (2 plus an extra for CP to operate properly) and one for a space, 4 characters or 28 columns will be occupied. This leaves 140 for the bar chart itself. It shall be necessary to scale the bar heights to 140 columns for the lower value in the bar (the number of meeting attendees), and to the number of columns remaining for the upper value in the bar (mailing list). This shall simply be 140 minus the column value for the meeting attendees.

In labelling the Y direction, we shall need tic marks every 10 out of 75 units. This translates to (10/75)*(140 columns) or every 18.67 columns. We shall approximate this with marks at column 28, 47, 65, 85, etc. increasing by 19, then 18, then 19, then 18, etc. until 168 is reached.

With labelling complete, the program goes as follows:

APPLICATION PROGRAM FOR: **HS**

Program	Comments
01♦LBL "PHILA"	
02 FIX 0	
03 "Y= MAILING LIST"	Header
04 "⊢, NUMBER"	
05 PRA	Lines
06 " ATTENDED"	
07 PRA	
08 "X= MEETING NO."	
09 PRA	
10 ADV	
11 26	
12 SKPCOL	
13 "0"	
14 ACA	
15 3	
16 SKPCHR	
17 "20"	Y direction
18 ACA	
19 23	Numeric
20 SKPCOL	
21 "40"	Labelling
22 ACA	
23 3	
24 SKPCHR	
25 "60"	
26 ACA	
27 14	
28 SKPCOL	
29 "75"	
30 ACA	
31 PRBUF	
32 27	
33 SKPCOL	
34 28	
35 ACCOL	
36 18	
37 SKPCOL	
38 28	
39 ACCOL	
40 1.003	
41 STO 01	
42♦LBL 01	
43 17	
44 SKPCOL	
45 28	
46 ACCOL	
47 18	Y direction
48 SKPCOL	
49 X<>Y	Tic marks
50 ACCOL	
51 ISG 01	
52 GTO 01	
53 9	
54 SKPCOL	
55 28	
56 ACCOL	
57 PRBUF	
58 1	Skip index for CP
59 STO 06	
60 1.016	Counter for 16 sets
61 STO 00	of inputs
62 FIX 0	
63♦LBL 00	
64 RCL 00	X label using CP
65 XROM "CP"	
66 1	
67 SKPCHR	Skip 1 space
68 77	
69 STO 03	Fill character for
70 140	lower bar
71 STO 04	Plot width
72 127	
73 STO 05	Fill column (lower)
74 "MTG "	
75 ARCL 00	Prompt for
76 "⊢?"	Meeting attendance
77 PROMPT	Value
78 75	
79 /	Scale the value
80 STO 02	
81 XROM "HS"	Call HS
82 61	
83 STO 03	Fill char. (upper)
84 28	
85 STO 05	Fill col. (upper)
86 "MAIL "	
87 ARCL 00	Prompting for
88 "⊢?"	Mailing list
89 PROMPT	Value
90 75	
91 /	Scale the value
92 RCL 02	Subtract from
93 -	meeting value
94 XROM "HS"	Call HS
95 PRBUF	
96 ISG 00	Repeat 16 times
97 GTO 00	
98 .END.	

Instructions for LBL PHILA:

Keystrokes	Display	Result
XEQ PHILA	MTG 1?	Prints header, prompts for meeting attendence input (from table 2 above)
18 R/S	MAIL 1?	Prompt for mailing list input
32 R/S	MTG 2?	Prints first bar and prompts for second meeting value
27 R/S	MAIL 2?	Prompt for 2nd mailing list input

30 R/S	MTG 3?	Prints 2nd bar and prompts for third meeting value
.	.	.
.	.	.
.	.	.
65 R/S	MTG 16?	Prints 15th bar and prompts for last meeting value
24 R/S	MAIL 16?	Prompt for last mailing list input
60 R/S	8	Prints last bar and stops

And finally, the bar chart would look like that in figure 2:

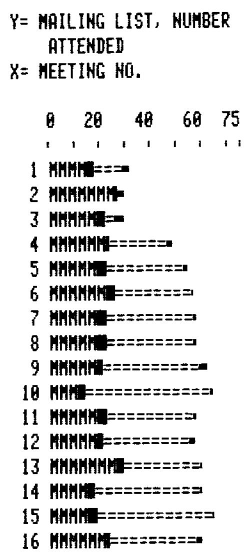


Figure 2. A bar chart of the mailing list size and the number of attendees of the 16 regular meetings of the Philadelphia Area PPC chapter.

COMPLETE INSTRUCTIONS FOR HS

Fill in the necessary data registers with the required information as follows:

R03 =	Fill character (#0 to 127 printer character)
R04 =	Plot width (0 to 168 columns)
R05 =	Fill column (0 to 127 ACCOL number)
X register =	Value to be plotted (between 0 and 1 inclusive)

Then XEQ HS and a bar will be accumulated into the print buffer. Remember that if additional information is to be added to the buffer, one must be aware of the buffer limits, as always. Failure to do so may cause overflow onto the next printed line. This includes the case of multiple HS bars being accumulated onto the same line, as in example 2 above.

<u>Limitations:</u> When a fill character is chosen, it is used to occupy groups of 7 full columns of the bar accumulated in the print buffer. However, all the characters in the printer character set actually only occupy the middle 5 columns of 7 allotted, leaving the outside columns blank, for clarity. A problem could have arisen if a bar's height was an exact multiple of 7 columns, and HS merely accumulated whole characters for that bar. The last character would leave its seventh column blank, and the height of the bar would be short by a single column. Therefore, a modification to the logic was made so that fill columns would be added up to the top of the bar, assuring acdurate height. Unfortuneately, this means that in some cases, as many as 7 fill columns have been placed at the top of a bar. If the fill column chosen does not appear to be a smooth extension of the fill character, then bars may look awkward at their tops. See "HS, HA and the Standard Printer Character Set", following the final page of this writeup for additional discussion of the limitations of HS

MORE EXAMPLES OF HS

<u>Example 3.</u> Produce a program which generates a histogram bar chart of class examination grades. The exam grades always range from 0 to 100 points. Let each bar represent the percentage of students who scored in a specific 10 point grade range. Maximum height of any bar should represent 50 percent of the population of the class. If a bar must exceed the 50% level, add an asterisk above the bar to indicate so.

The scores to be charted are shown in table 3 :

Student No.	Exam Score
1	70
2	80
3	32
4	75
5	76
6	89
7	95
8	62
9	100
10	79
11	74
12	81
13	79
14	77
15	73
16	51
17	76
18	65
19	78
20	74

Table 3. Exam scores as input to histogram routine TSTPLT.

We shall make the maximum height of a bar exactly 10 printer characters, or 70 columns. Character number 10 (the diamond) shall be the fill character and ACCOL number 8 (middle dot on) the fill column. After drawing the graph, the mean and standard deviation of the exam scores shall be printed.

We shall make this program interactive so the teacher is prompted to enter the exam grades, and then the histogram is generated. The register usage is as follows:

R00 = Input exam score
R01 = Counter for plotting
R02 = Used for X axis labelling
R03 = 10 (Fill character)
R04 = 70 (Plot width)
R05 = 8 (Fill column)
R06 to R11 = Statistical registers
R12 = 0 to 9 score totals
R13 = 10 to 19 score totals
. .
. .
. .
R21 = 90 to 99 score totals
R22 = 100 score totals

The TSTPLT program listing:

APPLICATION PROGRAM FOR: **HS**

```
01♦LBL "TSTPLT"
02 ΣREG 06
03 10
04 STO 03            Fill character
05 71
06 STO 04            Plot width
07 8
08 STO 05            Fill column
09♦LBL a
10 6.022             Clear registers
11 XROM "BC"         R06 - R22
12 "READY"
13 PROMPT
14♦LBL A             Input routine for
15 STO 00            entering scores
16 Σ+
17 RCL 00
18 10
19 /
20 12
21 +
22 1
23 ST+ IND Y         Increment the
24 CLA               appropriate register
25 FIX 0
26 ARCL 11
27 "⊢ = "
28 FIX 4
29 ARCL 00
30 PRA               Print entered value
```

```
31 RCL 11
32 RTN
33♦LBL B                Histogram plotting
34 ADV                   routine
35 ADV
36 "5% PER DIAMOND"
37 PRA
38 40                    Print the
39 ACCHR                 header
40 " MAX. = 50% "        information
41 ACA
42 41
43 ACCHR
44 PRBUF
45 12.022
46 STO 01
47 -1                    Numeric
48 STO 02                counters
49♦LBL 00
50 RCL 02
51 1
52 +
53 " "
54 FIX 0                 Bar printing
55 ARCL X                loop
56 100
57 X=Y?                  If label is 100
58 "⊢  "                 then add space
59 X=Y?
60 GTO 03
61 "⊢-"
62 RDN
63 9
64 +
65 STO 02
66 ARCL X
67 "⊢ "
68 10
69 X>Y?
70 "⊢  "
71♦LBL 03
72 ACA
73 RCL IND 01
74 XEQ "CODE"            Call bar plot rout.
75 ISG 01
76 GTO 00
77 FS? 00
78 " * = > 50%"          Asterisk if bar is
79 FS?C 00               greater than 50%
80 PRA
81 ADV
82 FIX 4
83 MEAN
84 "MEAN = "
85 ARCL X                Print mean
86 PRA                   and
87 ADV                   standard deviation
88 SDEV
89 "S.D. = "
90 ARCL X
91 PRA
92 XROM "PO"             Advance paper out
93 RTN
94♦LBL "CODE"           Bar plot routine
95 RCL 11
96 2
97 /
98 /
99 1
100 X<>Y
101 X>Y?
102 SF 01
103 FS? 01
104 SF 00
105 XROM "HS"
```

Program	Comments
106 FS? 01 107 " *" 108 FS?C 01 109 ACA 110 PRBUF 111 END	Add asterisk

We initialize by clearing registers 6 through 22 with the **BC** routine, and load in the input to **HS** . Then, the user is prompted 'READY' for test scores. After all scores have been entered, the histogram is printed, along with the mean and standard deviation:

Keystrokes	Display	Result
XEQ 'TSTPLT	READY	Initializes registers, clears R06-R22
1st score, XEQ A	1.0000	First score in, prints value
2nd score, XEQ A	2.0000	2nd score in, printed
. . .	. . .	. . .
Nth score, XEQ A	N.0000	Last score in, printed
XEQ B		Prints histogram, mean, and standard deviation
To begin again, XEQ a	READY	Initializes, etc.
1st score, XEQ A	1.0000	First score in, prints value

After all the scores have been entered and printed, the histogram in figure 3 is produced.

```
1 = 70.0000
2 = 80.0000
3 = 32.0000
4 = 75.0000
5 = 76.0000
6 = 89.0000
7 = 95.0000
8 = 62.0000
9 = 100.0000
10 = 79.0000
11 = 74.0000
12 = 81.0000
13 = 79.0000
14 = 77.0000
15 = 73.0000
16 = 51.0000
17 = 76.0000
18 = 65.0000
19 = 78.0000
20 = 74.0000
```

```
5% PER DIAMOND
( MAX. = 50% )
 0-9
 10-19
 20-29
 30-39 -
 40-49
 50-59 -
 60-69 ♦-
 70-79 ♦♦♦♦♦♦♦♦♦♦♦ *
 80-89 ♦♦-
 90-99 -
 100   -
 * = > 50%

MEAN = 74.0500

S.D. = 14.4093
```

Figure 3. A histogram of class grades entered into program TSTPLT from table 3 above.

The original goal of the histogram plot here was to have each single diamond character in a bar represent 5 percent of the total value of the student population. Thus, if the maximum height of a column could represent 50 percent, then a 70 column maximum height would assure 5 percent per fill character. However, because the last full 7 columns would be made up of fill columns since the tenth diamond wouldn't quite reach the 70th position, this goal couldn't be met. (See the limitation discussion above.) In order to assure a 10 diamond column for a full column, the plot width was made to be 71 columns. Then, **HS** would fill it with ten complete diamond characters plus a single additional fill column of ACCOL 8.

This program was submitted by Jack Sutton (5622) during the documentation phase of the **PPC ROM** project.

FURTHER DISCUSSION OF HS

Vertical Character Accumulation. This routine, originally submitted for inclusion in the **PPC ROM**, was written by Cliff Carrie (834). It is extremely useful for labelling the X direction of histograms, bar charts or any plots on the 82143A printer. Merely key in a number between 0 and 99 inclusive and the 2 digits will be accumulated into the print buffer as 5 ACCOL columns. If flag 12 is set when ACV is called, then the digits become twice as tall. The routine ACV listing:

BAR CODE ON PAGE 479

APPLICATION PROGRAM FOR: HS	
01♦LBL "ACV" 02 10 03 / 04 ENTER↑ 05 FRC 06 10 07 *	Separate into first and second digits
08 XEQ IND Y	Get 1st digit code
09 XEQ IND Y	Get 2nd digit code

```
10♦LBL 14
11 10
12 *
13 FRC
14 LASTX        Combine two codes
15 INT          to create ACCOL
16 16           values and accum-
17 *            ulate them into
18 RCL Z        print buffer
19 10
20 *
21 +
22 ACCOL
23 FRC
24 X<>Y
25 X≠0?
26 GTO 14
27 RTN          Codes for digits:
28♦LBL 00      0
29 .25552
30 RTN
31♦LBL 01      1
32 .22232
33 RTN
34♦LBL 02      2
35 .72452
36 RTN
37♦LBL 03      3
38 .34243
39 RTN
40♦LBL 04      4
41 .47564
42 RTN
43♦LBL 05      5
44 .34317
45 RTN
46♦LBL 06      6
47 .25316
48 RTN
49♦LBL 07      7
50 .22247
51 RTN
52♦LBL 08      8
53 .25252
54 RTN
55♦LBL 09      9
56 .34652
57 END
```

The barcode for routine ACV appears in Appendix N.

Example 4. Plot the following function: Y = EXP(-(X^2)/100) using HS . Let X values range from 0 to 20, in increments of 1. Label the columns using routine ACV. Y limits shall be from 0 to 1 inclusive.

Since ACV only occupies 5 printer columns, let us use 155 for the plot width and print the X labels double width (10 colums). We can choose printer symbol number 127 for a fill character and ACCOL number 127 for a fill column:

APPLICATION PROGRAM FOR: **HS**

```
01♦LBL "PLOT"
02 .02
03 STO 00       Numeric counter
04 155
05 STO 04       Plot width
06 127
07 STO 03       Fill character
08 STO 05       Fill column
09♦LBL 00
10 RCL 00       X label value
11 SF 12        Set double width
12 XEQ "ACV"    Accumulate label
13 CF 12
14 2
15 SKPCOL
16 RCL 00       Compute Y height
17 X↑2          of bar
18 1 E2
19 /
20 CHS
21 E↑X
22 XROM "HS"    Call HS
23 PRBUF        Print buffer
24 ISG 00
25 GTO 00
26 END
```

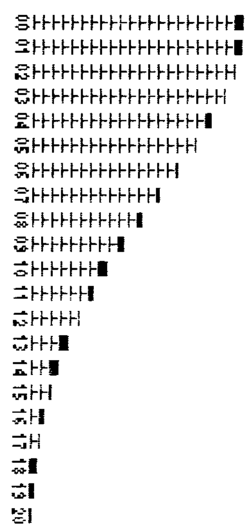


Figure 4. Plot of the function in Example 4, using the ACV routine to accumulate X labels, and using HS to produce histogram bars.

Routine Listing For: **HS**

```
48♦LBL "HS"      62 DSE Y
49 RCL 04        63 GTO 01
50 *             64 RDN
51 LASTX         65 INT
52 X>Y?          66 8
53 X<>Y          67 +
54 INT           68 RCL 05
55 7 E-5         69 GTO 00
56 +             70♦LBL 02
57 RCL 03        71 ACCOL
58 GTO 00        72♦LBL 00
59♦LBL 01        73 DSE Y
60 ACCHR         74 GTO 02
61♦LBL 00        75 RTN
```

LINE BY LINE ANALYSIS OF HS

Lines 49 and 50 calculate the height of the bar in printer columns.

Lines 51 to 53 test for overflow and substitute '1' for values that are too large.

Lines 54 through 58 set the X value up for acumulating fill characters and fill columns into the print buffer, based on the column value in X.

Lines 59 and 60 add a fill character to the print buffer.

Lines 61 through 69 decrement the column value by 7 columns for each fill character and move control to label 02 if there are 7 or fewer columns left to be accumulated.

Lines 70 through 75 accumulate fill columns up to the full height of the plot value.

REFERENCES FOR HS

See PPC Calculator Journal V7N2P5, V7N10P11.

CONTRIBUTORS HISTORY FOR HS

The original version of the high resolution histogram plotting routine was written by Ron Gordon (3449). Assistance was obtained from Cliff Carrie (834) and Bill Hermanson (4115) in implementing additional features.

FINAL REMARKS FOR HS

Many more possible applications exist for HS than can be described on these few pages. Further experimentation with various fill-characters and bar heights will reveal the large number of ways character-filled bars may be used to enhance a graph, provide a partition between tabular data lists, etc.

FURTHER ASSISTANCE ON HS

Contact Jake Schwartz (1820) at 7700 Fairfield St., Phila., Penna. 19152 (home phone 215-331-5324); or Cliff Carrie (834) at 152 Beverley Ave., Mount Royal, Quebec, Canada H3P1K7 (home phone 514-733-4866).

TECHNICAL DETAILS

XROM: 20,26 — **HS** — SIZE: 006

Stack Usage:
- 0 T: USED
- 1 Z: USED
- 2 Y: USED
- 3 X: USED
- 4 L: USED

Alpha Register Usage:
- 5 M:
- 6 N: NONE USED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q:
- 10 ⊢:
- 11 a: NONE USED
- 12 b:
- 13 c:
- 14 d:
- 15 e:

ΣREG: NOT USED

Data Registers:
- R00: NOT USED
- R06: FILL CHARACTER
- R07: PLOT WIDTH
- R08: FILL COLUMN
- R09: NOT USED
- R10: NOT USED
- R11: NOT USED
- R12: NOT USED

Flag Usage: NONE
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10:
- 25:

Display Mode: ANY

Angular Mode: NOT USED

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine:
00 (twice), 01, 02

Execution Time:
(5X + 1) seconds for X input from 0 to 1

Peripherals Required: 82143A Printer

Interruptible?	YES
Execute Anytime?	NO
Program File:	LG
Bytes In RAM:	40
Registers To Copy:	29

Other Comments:
This routine loads the print buffer, but does not PRBUF. Flags 12 and 13 must be clear to run HS .

HS, HA and The Standard Printer Character Set.

Routines HS and HA allow the user to choose a character from the standard printer character set for use in building histogram or bar charts. The set is numbered 0 through 127, and is accessed normally by the ACCHR (accumulate character) function. For HS and HA, the selected character number is stored into register R03. The set is shown in table 1.

Remember that these characters are utilized to fill 7-column portions of the histogram bars, even though the actual characters occupy the middle 5 characters of the 7 by 7 dot matrix. If a bar is to be an integral multiple of 7 columns high, HS or HA cannot simply accumulate that number of printer characters into the buffer to represent the numeric value, since the bar would then fall one column short of its correct height. To alleviate this problem, the routines substitute the individual fill-columns of dots for the last 7 columns in the bar.

0.	◆	43.	+	86.	V
1.	×	44.	,	87.	W
2.	x̄	45.	-	88.	X
3.	←	46.	.	89.	Y
4.	α	47.	/	90.	Z
5.	β	48.	0	91.	[
6.	Γ	49.	1	92.	\
7.	↓	50.	2	93.	]
8.	Δ	51.	3	94.	↑
9.	σ	52.	4	95.	_
10.	◆	53.	5	96.	`
11.	λ	54.	6	97.	a
12.	μ	55.	7	98.	b
13.	∡	56.	8	99.	c
14.	τ	57.	9	100.	d
15.	Φ	58.	:	101.	e
16.	Θ	59.	;	102.	f
17.	Ω	60.	<	103.	g
18.	δ	61.	=	104.	h
19.	Å	62.	>	105.	i
20.	å	63.	?	106.	j
21.	Ä	64.	@	107.	k
22.	ä	65.	A	108.	l
23.	Ö	66.	B	109.	m
24.	ö	67.	C	110.	n
25.	Ü	68.	D	111.	o
26.	ü	69.	E	112.	p
27.	Æ	70.	F	113.	q
28.	æ	71.	G	114.	r
29.	≠	72.	H	115.	s
30.	£	73.	I	116.	t
31.	▒	74.	J	117.	u
32.		75.	K	118.	v
33.	!	76.	L	119.	w
34.	"	77.	M	120.	x
35.	#	78.	N	121.	y
36.	$	79.	O	122.	z
37.	%	80.	P	123.	π
38.	&	81.	Q	124.	|
39.	'	82.	R	125.	→
40.	(	83.	S	126.	Σ
41.	)	84.	T	127.	⊢
42.	*	85.	U		

Table 1. The standard HP82143A printer character set.

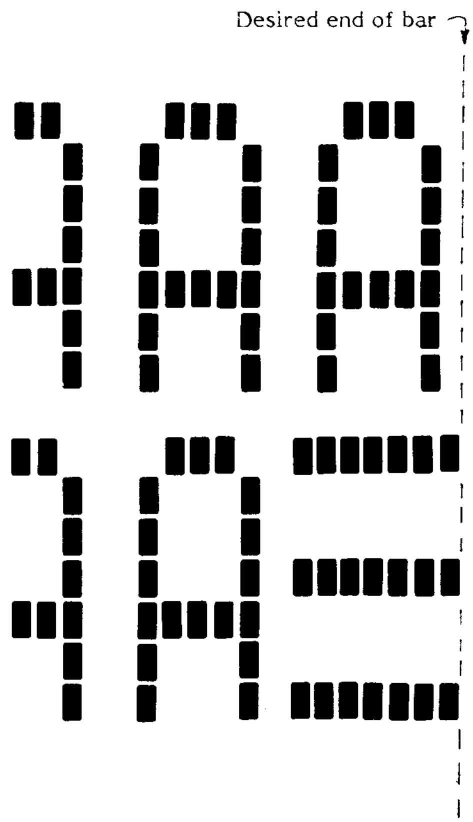


Figure 1. The right-hand end of a bar chart bar constructed by a row of 'A's (ACCHR #65), and a bar with the last 7 columns filled by ACCOL #73 fill-columns. The upper bar does not reach all the way to the desired column since printer characters do not occupy the end columns of the dot matrix. HS and HA automatically substitute fill-columns (the value stored in register R05) for the last 1 to 7 columns in a bar wherever necessary.

IF - INVERT FLAG

IF will invert the state of any of the 56 flags, making most of the historic Bug 3 capability available to all users. As shown in examples, IF can be put to practical uses, such as controlling Catalog 2 and 3 viewing, and increasing the speed of programs using the printer. IF is not needed for flags 0 to 29 and flag 48 (alpha mode).

Example 1: Use the BAT annunciator as an indicator of a running program's status, or as a special cue for keyboard input. It remains visible while entering data and, unlike the other annunciators, has absolutely no effect on mainframe functions. Cards cannot be read while it is on.

To merely invert the BAT status (controlled by flag 49) from off to on (or on to off) two program lines suffice: 49, XEQ IF. Note that both X and Y contents are left the same as they were before these two lines.

To turn BAT on from either state another line is needed: 49, FC?49, XEQ IF. (Stack contents now depend on whether IF was executed or not.) Similar program lines can turn BAT off from either state.

IF

For flag 49 the above lines work the same in a program or from the keyboard (not so for some flags). Admittedly if your program tends to leave BAT on for long periods you will experience a nagging curiosity about battery status, but if the card reader has worked recently there's no real reason for concern. Flag 49 can be cleared by turning the machine off and on, provided that the batteries are not actually low.

Example 2: A long-running program with only occasional printer operations can be sped up by manipulating flag 55. If the printer is connected, flag 55 (Printer existence) is automatically set at calculator turn-on and whenever control returns to the keyboard. Then, irrespective of flag 21 (Print enable) state or whether the printer is on or off, programs run slower, including sections with no intended involvement of the printer. IF can remedy this by clearing flag 55 for computation periods and then resetting Flag 55 just before printing.

Since NP has a variable running time it provides a good demonstration example for this. Fortunately IF preserves the two stack registers which NP needs. The following program NP1 prints NP results, and NP2 does the same but clears flag 55 for speed, using IF.

LBL "NP1"	LBL "NP2"	
VIEW Y	VIEW Y	
LBL 01	LBL 01	
XEQ NP	55	clears flag 55
VIEW X	XEQ IF	
x=y?	XEQ NP	
BEEP	55	sets flag 55
ST÷Y	XEQ IF	
GTO 01	VIEW X	
END	x=y?	
	BEEP	
	ST÷Y	
	GTO 01	
	END	

Running the Example 2 given in the NP write-up, NP1 takes 69 seconds, considerably longer than the original NP which took 45 seconds:

DO:	PRINTS:
Printer ON, MAN	
40,013,933 ENTER↑ 5001	
XEQ "NP1"	40,013,933 (at time = 0)
	5,309 (at time = 69 sec)
	7,537 (and then BEEPs)
R/S to stop the BEEPs	

The same procedure using NP2 saves 15 seconds. The comparison using different starting trial factors shows that if NP1 runs in less than 14 seconds NP2 takes longer:

Trial factor	5001	5101	5201	5251	5307
NP1 seconds	69	47	25	14	2
NP2 seconds	54	37	22	14	5

The investment is 8 program bytes (XEQ IF converts to XROM IF) and about 3 seconds for the two IF calls. Whenever a printing program has long no-printing periods, this is a good buy. If the program has stops, remember that flag 55 is reset when control returns to the keyboard. However, if you leave flag 55 clear, the calculator will set flag 21 along with 55 when the program stops. If you re-set flag 55, flag 21 will retain its status.

COMPLETE INSTRUCTIONS FOR IF

IF inverts the flag whose number is in the X register. Actually, as with the standard HP-41C flag functions, the integer part of the absolute value of X is used. Also, if X is 56 or greater it stops, showing DATA ERROR (...unless, unfortunately, flag 25 Error Ignore had been set beforehand). At completion of IF the original Y is left in X, and the original Z is left in Y and Z. T contains a duplicate of the new contents of the flag register d. The alpha registers are left clear, and L contains the number 12.

1) Key in the flag number (or in a program, place it in X).

2) XEQ IF. That's all, folks! Well, not really, since return to keyboard can reinvert flags 21, 51, 53, 54, or 55.

IF is useful in keyboard execution and as a subroutine, for inverting an individual system flag. If several flags are to be controlled it is generally better to generate an appropriate number or text and transfer it directly into the d register by STO d, X<>d or ASTO d. A program using the simple STO d technique always runs faster than one using IF, whether one or many flags are involved. IF usage is better when programming effort should be minimized, or when other flags' settings can't be pre-determined, but should remain unchanged. In a repetitive application one can use IF the first time, then by saving the d register contents using RCL d or SD one can use STO d or RD thereafter to reset the desired flags. The examples briefly introduce the system flag uses and effects, but much more is in the references, which are quite interesting.

MORE EXAMPLES OF IF

Example 3: Bug 3 is a characteristic of early production HP-41Cs, in which the system flags (30 to 55), not intended to be controllable by the flag functions, turned out to be controllable using the "indirect" feature, via a numbered register (not via the stack, though). The simple test for Bug 3 is:

Press Keys: 49, STO 00, SF IND 00
A Bug 3 machine will turn on the BAT annunciator.
A machine without Bug 3 displays NONEXISTENT.

Back in the olden days Bug 3 was used in pioneering experiments to probe the internal operation of the HP-41C. Generally those experiments can be repeated on a bug-less machine using **IF** with slightly different procedures (**IF** inverts, rather than sets or clears a flag, and uses X rather than a numbered register for control). However, **IF** and Bug 3 are different beasts, since **IF** runs only as a program, clears the alpha register and juggles the stack. Sometimes that is just too much - see Example 8.

The accompanying table lists the system flags (flags which except in Bug 3 machines cannot be set or cleared by the HP-41C Flag functions), notes some of their effects, and references the examples given here. The subject of system flag effects has not been fully documented, and different units behave differently, anyway, so experimentation and reading the references are recommended.

TABLE 1

System Flags	Some effects of setting by **IF**	Examples Useful	Examples Interesting
30 Catalog	Can run catalogs	Application Program 1	4
31 32 33 34 35 Peri-pherals	?		
36 8 Binary	FIX 10		
37 4 # of	(FIX A)		
38 2 digit	can		
39 1 display	display		
40 FIX	the		
41 ENG	goose		
42 GRAD			
43 RAD			
44 Cont. ON			
45 System Data Entry	Resume manual data entry	7	8
46 Partial Key Seq.	Major or minor per HP-41C version		
47 Shift	Next key has shifted function	5	
48 ALPHA	Better controlled by AON/AOFF		
49 Low BAT	BAT annunciator on No card reading	1, 3	
50 Message	Scroll VIEW Halt the goose	6	
51 SST			
52 PRGM	Insert garbage program lines		9
53 I/O	Clears immediately	FS? 53 skips one line	
54 Pause	Set PSE at return to keyboard		Line by Line Analysis
55 Printer Existence	Speed programs by clearing	2	

APPLICATION PROGRAM 1 FOR IF

CE (Catalog Entry) Using Flag 30

The program CE listed below permits entry at any point in CAT 2 (Application Modules and Peripherals) or CAT 3 (HP-41C functions). This has become an important item because of the huge number of global labels in external ROMs, the PPC ROM being a prime example. In CAT 2 the printer if present is always listed first, other ROMs following in port number order. Without the CE program it would be an impossibly long wait to see the catalog listing of a ROM put in a higher port number than the PPC ROM.

Catalog Entry Support Program

```
01 LBL "CE"
02 " "          Text consists of single SPACE
03 FS?03
04 CLA
05 XROM DC
06 RCL M
07 30
08 XROM  IF
09 STO P
10 "⊦READY"
11 AVIEW
12 END
```

1) Set flag 3 for CAT 3, clear flag 3 for CAT 2.
2) Key in the number of catalog items to skip.
3) XEQ "CE" (stops, showing READY, with flag 30 set)
4) R/S runs the catalog.

Lines 01 to 06 of this neat program generate the address of the desired catalog item, and lines 07 and 08 set the catalog flag. Lines 09 to 12 load the address into P which controls catalog running, then shift the address into place in P (and off the end of the alpha register part of P). They stop with alpha data in the display (if it were numeric, the P contents would get disturbed), ready for R/S to start the catalog. The leftmost nybble of P is the catalog number in binary form, and the three nybbles to the right of it are the binary representation of the catalog item number. All catalog numbers except 1 and 2 call up catalog 3, so if line 04 CLA gets executed, the first nybble is zero, getting catalog 3. Otherwise, the space (hex 20) calls catalog 2.

Example 4: CE can skip more items than the 117 in catalog 3, thus providing easy access to two of the "funny" catalogs which have aroused much interest in connection with ROM microcode investigations. Skipping 118 or 155 items in catalog 3 starts at the head of the two "funny" catalog segments, the latter one running well beyond 255, the maximum entry address CE handles.

These "catalog" items are now known to be strings of microcode read from addresses 1000 to 13FF in ROM 1 and are useful for deciphering two of the microcode bits which can't be read out by byte-jumping. If properly accessed they would name and implement the standard HP-41C functions, but in catalog readout each microcode value is read as a character, whether it was intended to be a character or not. The last part of the section (1389-13FF) consists of 118 microcode items which are scanned in running the proper catalog 3. Each is the address of a function name in the rest of the above section.

The funny catalogs result when the catalog run goes beyond 13FF and reads the microcode in addressess 1400 to 2388 (the rest of ROM 1 and into ROM 2). What is found there is interpreted to be an address in the block 1000 to 13FF, and that address is the starting point for each character sequence seen.

There are 4096 items including separators in these catalogs. The two "funny" catalogs which CE accesses run nicely with no problems, but many of the others end in crashes. See the references for details before trying the others, but changing line 02 in CE to a single letter from A to O and clearing flag 3 gives access to all the rest of them.

Example 5: Setting flag 47 (shift) is useful in a program which stops for keyboard input from a shifted key. The references show how to use Bug 3 or X<>d techniques for improving the "Arithmetic Teacher" program in the Standard Pac. By setting flag 47 just before the program halts for input at step 4, the user can choose between + - x ÷ without bothering with the shift key. IF is another way to accomplish this (see end of Complete Instructions section).

Example 6: Clearing flag 50 (message) in a program while a previous VIEW or AVIEW is in effect will scroll it each time a label is executed, just like the goose. This serves a dual purpose, indicating where the program is working and indicating passage through labels. Adds some variety too.

Flag 50 can also be cleared by an HP-41 "bug". Set Flag 25, use VIEW or AVIEW to put the desired message in display, then divide by zero or do something else illegal. This clears flag 25 and flag 50, but the display is not cleared. The result is the same as for 50 XEQ **IF** --a scrolling message.

Example 7: Setting flag 45 (system data entry) re-enables adding data to the string just entered. This could be of use if the first part of an alpha or numeric string is known beforehand, since the program can enter that part, and stop (with flag 45 set) for manual completion of the input. For a numeric entry, the program part must be an integer string. Thus, to enter a number known to be slightly over 180 degrees:

```
01 LBL "R"
02 45
03 XEQ IF
04 180
05 STOP
06 etc.
```

When 180 is displayed, press .02 to get 180.02, or back-arrow (←) to edit it as desired, then R/S to proceed.

Note that **IF** is used above as a subroutine. Setting flag 45 by keyboard execution of **IF** is pretty useless, since the last data entry is always the 12 in line 23 of **IF** itself.

Example 8: An example of a Bug 3 experiment that **IF** can't do is a manual sequence setting flag 45 taken from *PPC CALCULATOR JOURNAL*, V6N8P6

```
45 STO 00 FIX 9
ALPHA : ASTO X (then press":" 24 times) Aoff
SF IND 00
press any digit, STO 01, RCL 01
```

Without knowing what Bug 3 really does here, clearly the alpha and X registers are both involved, besides R00 for flag 45 control. There is just too much going on for **IF** to be used directly.

However, using surgery on **IF** , read it into RAM, insert line 28 STOP and change the label to IF+, thus permitting a manual alpha entry after the last number entry in **IF** , before setting flag 45.

The procedure, modified for IF+ is then:

DO:	RESULT:
FIX 0 45 XEQ "IF+"	NNN (future register d contents Note A
ALPHA (then press ":" 24 times) Aoff	
R/S	sets flag 45
When it stops, press any digit	⊠ 3 ⊠ 3 ⊠ 3 ⊠ 3 ⊠ 3 Note B
STO 01, RCL 01	:::0⊠ Note C

Note A ASTO X has to be left out (future d must be in X)

Note B This is an NNN displayed as 10 digits. They could be back arrowed (←) and edited.

Note C The NNN "normalized" into alpha data, showing where some of it came from.

Presumably this is similar to the result obtained using Bug 3. (*PPC CALCULATOR JOURNAL*, V8N2P6, reports an authentic potpourri of Bug 3 fun.)

Example 9: Setting flag 52 (program) does turn on the PRGM mode. However, if while this flag is set a running program encounters a number data entry line, a strange thing happens. The machine programs itself, repeatedly inserting the first data element as program lines between following existing program lines to the nulls and stops, suggesting "TRY AGAIN"! If flag 25 was set it does try again, finishing by packing again.

Routine Listing For:	**IF**
01♦LBL "IF"	17 X<> d
02 ABS	18 FC?C IND]
03 24	19 SF IND]
04 +	20 X<> d
05 STO [	21 STO [
06 8	22 RDN
07 ST/ [	23 12
08 MOD	24 -
09 RCL d	25 SCI IND X
10 X<> [	26 ARCL X
11 INT	27 X<>]
12 SCI IND X	28 STO d
13 ARCL X	29 RDN
14 X<>Y	30 CLA
15 X<>]	31 RTN
16 X<> \	32 RTN

LINE BY LINE ANALYSIS OF IF

The specified flag number starts in X. For analysis, express the flag number (F) in terms of bytes (y) and bits (i), F = 8y+i, where i can be from 0 to 7.

Lines 01 to 11 end up with i in Y, 3+y in X, and the initial flag status from d placed intact in M.

Lines 12 and 13 shift the M contents 7+y bytes to the left into N and (usually) O registers. The byte of the original d which contains the specified flag bit is now in the left most byte of N.

Lines 14 to 17 put the N part of the original d into the flag register d, and the O part into N, leaving the number i in register O.

Lines 18-21 toggle the specified flag using the i^{th} flag control (0 to 7), then replace an interim flag configuration into d while moving the toggled d section into M, where it is properly mated to the section in N from which it had been sliced.

Lines 23 to 26 assemble the final d contents in O.

Lines 27 and 28 load the final d contents into d, and lines 29 to 31 clean up and return control to the keyboard or the calling program.

Line 32 is needed for the one case where flag 54 (pause) is set by manual execution of **IF** . When that flag is set, the next RTN or END which would normally return control to the keyboard is converted to a PSE, following which program execution resumes, instead of stopping. Without the second RTN the following program (**CB**) would then get executed after **IF** sets flag 54. A STOP or PROMPT will clear flag 54 and stop with no pause.

REFERENCES FOR IF

Original Bug 3 Simulator: V7N4P23b.*
Interim version toggling all 56 flags: V7N8P10b, V7N10P17.
Flag 30 Catalogs:
- Early investigations using display mode control: V6N5P13b, V6N5P28d, V7N4P25, V7N5P3.
- P register relation to catalog control: V7N8P27, V8N5P13a.
- Catalogs organization, Catalog/ROM addresses: V8N5P14a, V8N5P21c.
- The Goose (by display flags 28, 29, 36, to 41): V7N3P3a, V7N5P56a
- Flag 45 system data entry: V6N8P6
- Flag 46 Partial Key sequence: V7N2P29, V8N4P30d, V8N5P15a.
- Flag 47 Shift: V6N6P3a, V7N4P25.
- Flag 49 low BAT: V6N5P28.
- Flag 50 Message: V6N5P30.
- Flag 52 PRGM: V7N2P36b.
- Flag 55 Printer existence: V8N4P24, V8N5P20d.
- More on most of above items, with many examples: See page 79 of *SYNTHETIC PROGRAMMING*, by William C. Wickes

CONTRIBUTORS HISTORY FOR IF

Jon Doig (4318) wrote the original "Bug 3 Simulator" program for flags 8 to 55, reported in the first of the references. William C. Wickes (3735) provided an ultra short version for flags 24 to 55. Gerard Westen (4780) modified Doig's program so it could toggle all 56 flags and save y. Roger Hill (4940) revised it, using a different byte-shifting method, shortening the run time, saving both Y and Z, and adding several neat features. Roger Hill (4940) wrote the CE (Catalog Entry) program.

*Unspecified references are *PPC CALCULATOR JOURNAL*.

FURTHER ASSISTANCE ON IF

Call Les Matson (5608) at W- (617) 258-1764 or H- (617) 235-7955.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,49	IF	SIZE: 000

Stack Usage:
- 0 T: new d
- 1 Z: Z
- 2 Y: Z
- 3 X: Y
- 4 L: 12

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: RESTORED WITH INVERTED FLAG
- 15 e NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: ALL BUT DESIGNATED FLAGS ARE UNCHANGED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: 1.3 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	IF
Bytes In RAM:	56
Registers To Copy:	60

Other Comments:
Setting flag 30 (catalog) can result in crash.
Setting flag 46 (Partial sequence) can result in crash.
Setting flag 52 (PRGM) can intersperse garbage lines in the program.

IG - INTEGRATE

This routine uses the Romberg algorithm to calculate a numerical approximation of the definite integral of a function. The routine is iterative in that increasingly accurate approximations are calculated until two rounded consecutive approximations are equal. The routine is automatic in that no step-size information has to be provided. The desired accuracy of the final approximation is determined by the display setting. The consecutive approximations may be viewed by setting a flag.

Example 1: Use IG to approximate $\int_0^1 4/(x^2 + 1)\,dx$.

to five significant digits.

1. SIZE 030 minimum.
2. Select a display setting of SCI 4.
3. Set flag F10 to view the successive approximations
4. Key the integrand of the above integral as a function in program memory starting with a global label and ending with a RTN. Assume x is in the X register and leave f(x) in the X register. Key in the following steps for this example.

```
01*LBL "FX1"
02 X↑2
03 1
04 +
05 4
06 X<>Y
07 /
08 RTN
```

5. Alpha-store the global label name into register R10.
   Key "FX1" ASTO 10.
6. Enter the limits of integration into the stack as 0 ENTER↑ 1.
7. XEQ " IG ".

The following sequence of numbers will be displayed.

3.2000+00
3.1405+00
3.1413+00
3.1416+00
3.1416+00

This example takes about 49 seconds to run. The true answer is pi, and the displayed result is accurate to five significant digits. Switching to FIX 9 we see the last value returned is 3.141592651 but because we were in SCI 4 mode we should not expect more than 4 decimal places of accuracy. Displaying more digits will cause the program to run longer.

Calculate the same integral a second time but change to SCI 6 display mode. Since the function subroutine and the global label name in R10 are not changed, simply key 0 ENTER↑ 1 and XEQ " IG " again. The following sequence of numbers will be displayed.

3.200000+00
3.140464+00
3.141329+00
3.141598+00
3.141593+00
3.141593+00

The last value is returned after about 89 seconds.

COMPLETE INSTRUCTIONS FOR IG

(Keyboard Operation):

To calculate $\int_a^b f(x)dx$

1) Select SIZE. SIZE 030 is the recommended minimum. A few integrals may require a larger size.

2) Set display mode. The display setting will control the accuracy of the final approximation. In general, a display mode of SCI n will return a value correctly rounded to n+1 significant digits. Larger values of n will cause the program to run longer so it is best to select the minimum value of n that is acceptable. The use of SCI or ENG display modes are generally preferable to the FIX mode.

3) Select display view option. Flag 10 controls a display option. If F10 is set then the successive approximations that the program calculates will be displayed. In this manner the user may view the progress of the iterations. If F10 is set and a printer is connected the approximations will be printed. If F10 is clear only the final approximation is returned in the X-register.

4) Specify the integrand. The integrand represented by the function f(x) must be programmed as a subroutine in program memory which starts with a global label name and ends with a RTN or END instruction. This label name should be of six or less characters and will be stored in R10. The input x and the output f(x) are both assumed to be in the X-register. Since global label search begins at the bottom of program memory, when the f(x) subroutine is in RAM it should be located at the bottom of RAM. Using a single character global label name will reduce execution time. The f(x) program should not use registers R10-R29 or use flags F09 and F10.

5) Alpha-store the global label name from step 4 (six or less alpha characters) in R10.

6) Enter limits of integration. The lower and upper limits of integration, a and b, respectively, are to be keyed in as a ENTER↑ b so that a is keyed into the Y-register and b is keyed in the X-register.

7) Execute IG program. Key XEQ " IG ". Program execution will commence, and if F10 is set, the consecutive approximations will be displayed. If a printer is connected and turned on the approximations will be printed. The final approximation will be left in the X-register when the program ends.

MORE EXAMPLES OF IG

Example 2: Calculate $\int_0^1 x^{1/2}dx$

1. Select SIZE 030.
2. Select a display mode of SCI 4.
3. Set flag F10 to VIEW the approximations.
4. Key in the following routine for f(x).

```
LBL*FX2
SQRT
RTN
```

5. Key "FX2" in the alpha register and ASTO 10.
6. Key in the limits of integration as 0 ENTER↑ 1.
7. XEQ " IG ".

The following approximations will be displayed.

7.0711-01
6.6947-01
6.6667-01
6.6667-01

The final answer is returned after about 23 seconds.
The true answer is 2/3.

Example 3: Calculate $\int_0^1 \sin(\pi x)\,dx$

1. Select SIZE 030.
2. Select a display mode of SCI 4 and select RADIANS angle mode.
3. Set flag F10 to VIEW the approximations.
4. Key in the following routine for f(x).

```
LBL*FX3
  PI
  *
  SIN
  RTN
```

5. Key "FX3" in alpha and ASTO 10.
6. Key in the limits of integration as 0 ENTER↑ 1.
7. XEQ " IG ".
The following approximations will be displayed.

1.0000+00
6.0355-01
6.3789-01
6.3660-01
6.3662-01
6.3662-01

The final answer is returned after about 102 seconds.
The true answer is $2/\pi$.

In the following examples only the original problem and the numbers output are given.

Example 4: Calculate $\int_0^1 LN(x)\,dx$

-6.931-01
-9.331-01
-9.879-01
-9.972-01
-9.993-01
-9.998-01
-1.000+00
-1.000+00

The approximate time is 301 seconds in SCI 3.
The true answer is exactly -1.

Example 5: Calculate $\int_0^1 \frac{x^{1/2}}{x-1} - \frac{1}{LN(x)}\,dx$

2.8481-02
3.6106-02
3.6618-02
3.6519-02
3.6496-02
3.6491-02
3.6490-02
3.6490-02

The approximate time is 402 seconds in SCI 4.
The true answer to 7 decimals is 0.0364900.

Example 6: Calculate $\int_0^2 [x(4-x)]^{1/2}\,dx$

3.46410162
3.15270628
3.14152977
3.14159373
3.14159265
3.14159265

The approximate time is 85 seconds in FIX 8.
The true answer is π.

Example 7: Calculate $\int_0^{\pi} \frac{600*\sin^2(x)}{x^{1/2} + (x+600\pi)^{1/2}}\,dx$

Remember to use RADIANS angle mode.

4.21808+01
1.75899+01
2.13355+01
2.10986+01
2.11020+01
2.11020+01

The approximate time is 146 seconds in SCI 5.
The correct answer to 5 decimals is 21.10204

Example 8: Calculate $\int_0^1 COS(LN(x))\,dx$

Use RADIANS angle mode.

7.692-01
4.563-01
4.765-01
5.035-01
5.018-01
4.999-01
4.999-01

The approximate time is 213 seconds in SCI 3.
The true answer is exactly 0.5

Example 9: Calculate $\int_0^1 x^{-1/2}\,dx$

1.414+00
1.710+00
1.865+00
1.934+00
1.967+00
1.984+00
1.992+00
1.996+00
1.998+00
1.999+00
1.999+00

The approximate time is 33 minutes and 52 seconds in SCI 3. The true answer is exactly 2.

Example 10: Calculate $\int_0^1 (1-x^2)^{1/2}\,dx$

8.6602540-01
7.8817657-01
7.8538244-01
7.8539843-01
7.8539816-01
7.8539816-01

The approximate time is 85 seconds in SCI 7. The true answer is $\pi/4$.

Example 11: Calculate: $\int_{-1}^{1} \frac{x^7(1-x^2)^{1/2}}{(2-x)^{13/2}}\,dx$

0.0000+00
6.6870-03
3.0827-02
2.3585-02
2.3850-02
2.3857-02
2.3857-02

The approximate time is 342 seconds in SCI 4. The true answer to 7 decimal places is 0.0238566

Example 12: Calculate: $\int_{-1}^{1} [(1-x^2)(2-x)]^{1/2}\,dx$

2.8284+00
2.2239+00
2.2033+00
2.2033+00

The approximate time is 29 seconds in SCI 4. The true answer to 6 decimal places is 2.203345

Examples 4 to 8 above are taken from Reference 8 and are © Copyright 1980, Hewlett-Packard Company. Reproduced with permission.

FURTHER DISCUSSION OF **IG**

The ability of any numerical integrator to determine the definite integral of a function is basically determined by:

1. The behavior of the function over the interval of integration.
2. The selected numerical method.
3. The required accuracy of the solution.

Due to the above it is not possible to fully expand in this documentation on the applications of **IG** or to fully discuss the difficulties that may arise. This type of information must be obtained from books and publications on numerical methods such as those given in the References.

However, References 5 and 8 would probably be the most informative on the Romberg numerical integration method. It was from those two References that the **IG** routine was born. In Reference 8, the theory underlying the integrate function key on the HP-34C calculator is presented in quite some detail. Reference 5 describes the Romberg integration procedure, along with the theory of many other methods. An extensive bibliography on the subject is also provided in Reference 5.

Reference 11 provides highly valuable practical insight into the HP-34C integration key function and its applications.

Mathematical Background of **IG**

The method underlying the program is due to Romberg (Reference 1) and is essentially an application of Richardson's extrapolation procedure to the Euler-Maclaurin sum formula. Romberg was first to describe the method in recursive form. Commencing with improved midpoint rule estimates, the continued application of extrapolation to the limit produces a lower triangular matrix. Although the columns of the matrix converge to the solution, the diagonal elements converge asymptotically faster than any geometric series, or, superlinearly. Program shut-off occurs when two rounded consecutive diagonal elements are equal. The diagonal elements M(k,k) are the values displayed when flag F10 is set.

Assuming that the number of divisions of the interval of integration is increased by improved midpoint rule estimates, one would assume that convergence would occur when we made the number of intervals high enough. However, at some point, roundoff errors eventually dominate and our effective accuracy decreases. The Romberg method allows the simulation of a high number of sub-intervals or divisions which decreases the error, without actually increasing the number of sub-intervals. This process is called extrapolation to the limit.

Romberg integration is iterative, automatic, and non-adaptive. It is iterative in that it produces increasingly accurate estimates of the solution until the convergence criterion is satisfied. It is automatic in that the number of function evaluations depends upon the behavior of that function over the interval of integration. It is non-adaptive in that function evaluations occur at a fixed set of points, independent of the function.

As the Romberg method successively halves the interval of integration to produce improved midpoint rule estimates, it uses all previously computed functional evaluations at each stage. The retention of all previously calculated functional evaluations is a significant aspect of the Romberg algorithm. As the function is evaluated at the center of each interval, the end points of the intervals are not used as sample points. Hence the endpoints of the interval of integration, a and b are also not used as sample points. This allows certain improper integrals to be approximated. For example, LN(x) can be integrated over the interval (0, 1] even though LN(0) is undefined.

A refinement was implemented in the basic Romberg scheme. If uniformly spaced sample points are taken, periodic integrands may sometimes cause a problem due to resonance phenomena. In this program the sampling has been made non-uniform by a non-linear substitution. Such a substitution was implemented in the HP-34C calculator integration key routine.

A complete discussion of the theory of Romberg integration is given in References 2 and 3. Reference 8 provided the starting point for IG and users are urged to consult that work.

FORMULAS USED IN IG

$$(1) \quad I = \int_a^b f(x)\,dx$$

There are three steps to the solution of (1) using the Romberg method.

A. Change of limits:

The interval of integration [a,b] is changed to the interval [-1,1] by the change of variable:

Let x = [(b-a)/2]*t + (b+a)/2, then dx = [(b-a)/2]dt

After substitution into the right side of (1) and simplifying we have:

$$(2) \quad I = [(b-a)/2]* \int_{-1}^{1} f([(b-a)/2]*t + (b+a)/2)\,dt$$

B. Introduce non-uniform sample points:

Equation (2) is further refined by another change of variable which causes the sample points to be non-uniform over the original interval of integration.

Let $t = (3/2)*u - (1/2)*u^3$,

then $dt = (3/2)(1-u^2)du$

After substitution into the right side of equation (2) and some simplification we have:

$$(3) \quad I = 3(b-a)/4* \int_{-1}^{1} f([(b-a)/4]u(3-u^2) + (b+a)/2)(1-u^2)du$$

Now a uniform distribution of sample points in u over the interval [-1,1] will be transformed to a non-uniform distribution of sample points x over the original interval of integration [a,b]. The Romberg extrapolation procedure is applied next to produce elements of a matrix M(k,k).

C. Generating the Romberg Matrix

$$(4) \quad I = \lim_{k \to \infty} M(k,k) \qquad k=0,1,2,3,\ldots$$

where:

$$(5) \quad u_0 = -1 + 2^{-k}$$

$$(6) \quad u_i = u_{i-1} + 2^{1-k}$$

$$(7) \quad x_i = [(b-a)/4]u_i(3-u_i^2) + (b+a)/2$$

$$(8) \quad S_0 = f((a+b)/2)$$

$$(9) \quad S_k = \sum_{i=0}^{2^k-1} f(x_i)(1-u_i^2) + S_{k-1}$$

$$(10) \quad M(k,0) = [3(b-a)/4]*2^{-k}*S_k$$

and finally

$$(11) \quad M(k,j) = M(k,j-1) + \frac{[M(k,j-1) - M(k-1,j-1)]}{4^j - 1}$$

The elements M(k,j) form a lower triangular matrix as follows:

M(0,0)					
M(1,0)	M(1,1)				
M(2,0)	M(2,1)	M(2,2)			
M(3,0)	M(3,1)	M(3,2)	M(3,3)		
M(4,0)	M(4,1)	M(4,2)	M(4,3)	M(4,4)	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Equation (11) indicates that each element M(k,j) depends on the element immediately to its left, and on the element above the one immediately to its left. Only the most recent row M(k,0), M(k,1), M(k,2),... M(k,k) is stored in data registers R18 and up.

The program halts when two rounded consecutive diagonal elements M(k-1,k-1) and M(k,k) are equal.

When flag F10 is set, the first result to be displayed is (4/3)*M(0,0) which in fact is the element M(0,1). Subsequent displays show M(1,1), M(2,2), M(3,3), etc., until convergence occurs. M(0,0) is not displayed. The final display is M(k,k).

Analysis of a numerical example.

To further illustrate the Romberg method used in **IG**, the evaluation of the following function will be described in detail.

$$I = \int_0^1 (1-x^2)^{1/2}\,dx = \pi/4$$

$$= 7.853981635\text{-}01$$

The transformation from u to x is shown in Table 1, where linear samples in u, over the interval (-1,1), are transformed to non-linear samples in x over the interval (0,1). For k=0, the first point is u=0, at the center of the interval (-1,1). This then divides the interval in u into two new subintervals, (-1,0) and (0,1). The next iteration for k=1 samples u at the center of the new intervals at -1/2 and 1/2.

k	u	x
0	0	0.5
1	-1/2, 1/2	0.15625, 0.84375
2	-3/4, -1/4 1/4, 3/4	0.04297, 0.31641 0.68359, 0.95703
3	-7/8, -5/8, -3/8, -1/8, 1/8, 3/8, 5/8, 7/8	0.01123, 0.09229, 0.23193, 0.40674, 0.59326, 0.76807, 0.90771, 0.98877

TABLE 1
Transformation of u into x

k	M(k,0)	M(k,1)	M(k,2)	M(k,3)
0	0.64951905			
1	0.75351219	0.78817657		
2	0.77754585	0.78555708	0.78538244	
3	0.78344255	0.78540811	0.78539818	0.78539843
4	0.78490973	0.78539879	0.78539816	0.78539816
	M(4,4)=0.78539816			

TABLE 2
M Matrix Generation

This process is continued, with u being sampled at the center of each new interval. The sample point in u is then transformed to a sample point in x by equation (7) as described previously.

The evaluation of the M matrix is shown in TABLE 2. The elements M(k,0) are the improved midpoint rule estimates; the elements M(k,1) represent the first extrapolation, and in fact turn out to be the values obtained using Simpson's Rule improvements; the elements M(k,2) represent the second extrapolation, and in fact turn out to be the values obtained using the closed form, Newton-Cotes formula for five points.

In this program the element M(0,1) is the first element displayed if flag F10 is set, but is not used for any later calculations. M(0,0) is not displayed.

When FIX 6, SCI 5, or ENG 5 display modes are used in this example, the program halts after iteration k=4 since M(3,3) and M(4,4) agree to six significant digits. The element M(4,4) is returned to the X-register as the final solution.

Convergence of **IG**.

Convergence occurs, and execution halts, when two consecutive rounded diagonal elements, M(k,k), are equal. An advantage of the automatic Romberg integrator is that no decision has to be made in advance concerning the optimum step size. The convergence criterion of **IG** is not as strict as that implemented in the HP-34C. In the HP-34C the program does not halt until three consecutive diagonal elements agree to the desired accuracy. Due to limited space, this criterion could not be implemented in the **PPC ROM**. For most integrals this difference will not be noticed, but it is possible that a few integrals evaluated by **IG** will halt prematurely.

Timing Data

IG execution time is mainly proportional to the number of times the f(x) subroutine is called. However, it is also proportional to the execution time of the f(x) routine, and the number of characters in the global label name of the f(x) routine. It is also governed by the location of the f(x) routine global label in program memory. The further the global label is from the bottom end of program memroy, the longer the execution time. At the end of the kth iteration, the function f(x) will have been called

$$2^{k+1} - 1$$

times. The time for k iterations in seconds is given approximately by:

$$T_k = 1 + 3.4K + (0.735 + t_l + t_f)*(2^{k+1} - 1)$$

where:

t_l = time to search and locate the f(x) global label.

t_f = time to execute the f(x) routine.

Each new iteration takes as long as all previous iterations. SIZE 030 will allow iterations up to k=11. The minimum time to complete this many iterations is approximately one hour.

The execution times given for all the previous numerical examples were obtained using a calculator which executed 500 "+" instructions in 15 seconds.

Routine Listing For: IG

```
01♦LBL "IG"        46 +
02♦LBL B           47 X<Y?
03 STO 17          48 GTO 02
04 X<>Y            49 RCL 11
05 -               50 STO 13
06 4               51 18
07 /               52 STO 12
08 STO 16          53 E
09 ST- 17          54 ST+ 11
10 ST- 17          55 RCL 15
11 .               56 RCL 16
12 STO 15          57 1.5
13 STO 11          58 *
14 STO 18          59 *
15 SF 09           60 RCL 14
16♦LBL 01          61 *
17 E               62♦LBL 03
18 2               63 R↑
19 STO 14          64 4
20 RCL 11          65 *
21 CHS             66 ENTER↑
22 Y↑X             67 DSE Y
23 ST* 14          68 X<> Z
24 E               69 ENTER↑
25 -               70 X<> IND 12
26♦LBL 02          71 ST- Y
27 STO 12          72 RND
28 X↑2             73 X<> Z
29 -               74 /
30 STO 13          75 RCL IND 12
31 2               76 +
32 +               77 ISG 12
33 RCL 12          78 STOP
34 *               79 DSE 13
35 RCL 16          80 GTO 03
36 *               81 STO IND 12
37 RCL 17          82 FS? 10
38 +               83 VIEW X
39 XEQ IND 10      84 FS?C 09
40 RCL 13          85 GTO 01
41 *               86 RND
42 ST+ 15          87 X≠Y?
43 E               88 GTO 01
44 RCL 12          89 LASTX
45 RCL 14          90 RTN
```

LINE BY LINE ANALYSIS OF IG

Lines 01-10 intialize the program by storing the constants (b-a)/4 and (b+a)/2 in R16 and R17 respectively.

Lines 11-14 initialize S_k, k, and M(k,k) for k=0.

Line 15 is used to force the program to run through at least two iterations. See also lines 84 and 85.

Lines 16-25 calculate u_0 and the step size 2^{k-1}.

Lines 26-48 calculate x_i (line 38), $f(x_i)$, (line 39), and S_k (line 42).

Lines 49-61 calculate M(k,0).

Lines 62-80 calculate M(k,j).

Lines 86-88 are the exit test. The routine ends when two consecutive rounded approximations are equal.

Lines 89-90 recall the final approximation and halt.

REFERENCES FOR IG

1. W. Romberg, "Vereinfachte Numerische Integration," NORSKE VIDENSKAB. SELSKAB., FORH. (TRONDHEIM) 28, No. 7, 1955.
2. F.L. Bauer, H Rutishauser, E.L. Stiefel, "New Aspects in Numerical Quadrature," in EXPERIMENTAL ARITHMETIC, HIGH SPEED COMPUTING AND MATHEMATICS, pp. 199-218, American Mathematical Society, Providence, Rhode Island, 1963.
3. E.L. Stiefel, AN INTRODUCTION TO NUMERICAL MATHEMATICS, Academic Press, New York, 1963.
4. Abramowitz and Stegun, HANDBOOK OF MATHEMATICAL FUNCTIONS, National Bureau of Standards, Applied Mathematics Series, No. 55, 1964
5. P. J. Davis and P. Rabinowitz, METHODS OF NUMERICAL INTEGRATION, Academic Press, New York, 1975.
6. John Kennedy, PPC Journal, V6N5P18, August 1977.
7. Irving Allen Dodes, NUMERICAL ANALYSIS FOR COMPUTER SCIENCE, North-Holland, New York, 1978.
8. William M. Kahan, "Handheld Calculator Evaluates Integrals," Hewlett-Packard Journal, August 1980.
9. Read Predmore, PPC CALCULATOR JOURNAL, V7N6P4, July-August 1980.
10. Read Predmore, PPC CALCULATOR JOURNAL, V7N9P13, November 1980.
11. OWNER'S HANDBOOK AND PROGRAMMING GUIDE, Hewlett-Packard HP-34C Calculator, Hewlett-Packard Company.

CONTRIBUTORS HISTORY FOR IG

Several numerical integration methods are in popular use today. References 5 and 8 provide insight into the performance limitations of some of these methods. Other methods may be faster than the iterative Romberg method, but the Romberg method is very efficient in that it uses all previously calculated estimates and it is automatic in the sense that step-size information does not have to be provided. Also, error propogation is reduced by the matrix calculation procedure which in turn greatly speeds convergence.

Using References 5 and 8 Read Predmore (5184) produced a very efficient, compact, and elegant HP-41C program (Reference 10) using the iterative Romberg method. That program was almost identical in function to the routine underlying the integrate key on the HP-34C calculator as described in Reference 8. The Reference 10 program formed the basis for IG.

John Kennedy (918) reduced the program size to its final form. Harry Bertucelli (3994) suggested register usage to allow IG to be used with SV. The documentation on IG was written by Graeme Dennes (1757) after proof reading by Read Predmore (5184). Thanks to the Hewlett-Packard Company for allowing the reproduction of numerical examples from Reference 8.

FINAL REMARKS FOR IG

By adding some enhancements IG could be improved to make the procedure more closely fit the complete method used in the HP-34C, the first calculator to have an integration routine as a built-in function. Speed is also an area of needed improvement.

FURTHER ASSISTANCE ON IG

Read Predmore (5184) phone: (413) 367-9513

Graeme Dennes (1757) phone (415) 592-2957 evenings

NOTES

TECHNICAL DETAILS

XROM: 20, 09 | **IG** | SIZE: 030 minimum

Stack Usage:

- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:

- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:

- 9 Q: not used
- 10 ⊦: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:

- R10: function LBL name
- R11: k = counter
- R12: u_i
- R13: $1 - u_i^2$
- R14: delta u = 2^{1-k}
- R15: S_k
- R16: (b-a)/4
- R17: (b+a)/2
- R18: M(k,0)
- R19: M(k,1)
- R20: M(k,2)
- ⋮ ⋮

Flag Usage:

- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: used to force two iterations
- 10: set to display approximations
- 25: not used

Display Mode:

SCI n recommended

Angular Mode:

not used, but may be required by function

Unused Subroutine Levels:

4

Global Labels Called:

Direct	Secondary
function LBL in R10	

Local Labels In This Routine:

B, 01, 02, 03

Execution Time: see IG documentation for detailed timing information.

Peripherals Required: none (printer recommended)

Interruptible? yes

Execute Anytime? no

Program File: IG

Bytes In RAM: 131

Registers To Copy: 43

Other Comments:

APPENDIX G CONTINUED FROM PAGE 187.

BUG 7: Fragmented seven character alpha strings are placed into X with BUG 7 machines. Seventh character from the same column of the HEX table will compare equal if the first six characters are the same. See PPC J, V6N8P23b.

BUG 8: This is a non-compile if OFF in PRGM mode bug. Editing a program and turning the HP-41 off while in PRGM mode will leave the program compiled as prior to editing. See PPC J, V6N8P23c.

BUG 9: The Catalog BUG is a bug found in all HP-41's. The program pointer may be placed into the assignment registers using bug 9 by stopping a catalog and deleting lines. For details see PPC CJ, V7N9P25a.

C

CHARACTER - A display character is what the user sees in one of the twelve positions in the HP-41C display. A memory character consists of a single byte in the text portion of a string.

CODE- The name of a program written by Bill Wickes (3735) that translated HEX codes in ALPHA into the bytes of the HP-41 placing them in the X register. PPC ROM routine **HN** performs this task.

COMPILE - A computer term that describes the operation of determining the location of a branch destination and placing the distance value in the instruction. An HP-41 GTO 01 will take longer to get to LBL 01 (within range) the first time, but will run faster the second time, because the number of bytes is "compiled" and stored in the GTO instruction. Also see BUG 6.

CURTAIN - The dividing line between data register and program memory. Its value is a pointer to register 00 maintained in Register c, as an absolute register number. The curtain is moved throughout memory by changing this value using synthetic instructions. See Appendix M on Curtain Moving.

D

DECODE - The name of a program written by Bill Wickes (3735) that translated the seven bytes (NNN) of the X register into the HEX codes of the HP-41. This is done by the **NH** ROM routine.

DEEP SLEEP - An HP term that describes one of three states of the HP-41C/CV. Deep sleep is the "OFF" state of minimum power, typically a few microamperes. Also see light sleep and Run Mode.

F

FULL MAN - The character displayed by viewing byte 01 (HEX). See Man Characters herein.

G

GLOBAL LABEL - A label consisting of any one through seven ALPHA characters (other than comma, period, or colon) except that the single characters A through J and a through e which are Local Labels. A Global Label may be addressed from anywhere in memory.

H

HEX - Hexadecimal, pertaining to base 16. HP-41 instructions are grouped in a table of 16 columns and 16 rows. These instructions are often identified by two HEX characters--row, column.

DECIMAL	HEX	DECIMAL	HEX
0-9	0-9	13	D
10	A	14	E
11	B	15	F
12	C		

HEX TABLE - A table of 256 values (for the HP-41) arranged 16 x 16 and identified by counting in HEXadecimal. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Two HEX characters, row and column, identify a particular instruction. The HEX value is especially useful, because the actual BINARY value may be conveniently derived from it. Also see HEX.

L

L REGISTER - The fifth register (004 absolute) of HP-41 memory. The L register is called the LAST X register and is not part of the Stack, but actively interacts with the X register. Also see Status Registers.

LIGHT SLEEP - An HP term that describes the ON, but not running mode of the HP-41C. The HP-41 display is operating and the keyboard is "alive". Also see Deep Sleep and Run Mode.

M

M REGISTER - The sixth register (005 absolute) of HP-41 memory--the right most seven bytes of the ALPHA register. M is one of 16 status registers (R00 through R15 absolute) recorded during a WSTS operation. M may be used for general scratch and non-normalizing STO/RCL. Also see ALPHA Register.

MAN CHARACTERS - The characters represented by hex codes 01 (full man, ᛡ), 04 (one-armed man, ᛣ), 03 (armless man, ᛦ), and 06 (one-leg man, ᛇ). It is speculated that HP intended these characters for a hangman game that was never implemented.

MASTER CLEAR - Key sequence on HP-41 to clear all memory, set certain flags and SIZE to provide 46 4/7 program registers. While pressing the back arrow key, press and release the ON key. Release ' backarrow key. This sequence results in MEMORY LOST.

MEMORY LOST - An HP-41 display message that results when the cold start constant in the c register is not HEX 169. All program and data memory are cleared. A Master Clear, 0 STO c, or curtain directly above a Nonexistent register will produce MEMORY LOST.

MEMORY VOID - 176 registers between the 16 status registers and first key assignment are not addressed by the 41 operating system. In absolute registers 0 through 15 are status, 16 through 191 are the void, 192 through 511 are program/data memory. The 41 operating system automatically "jumps" the void.

MICROCODE - The sequence of assembly language operations performed by the HP-41C micro processor is

APPENDIX G CONTINUED ON PAGE 233.

IP - INITIALIZE PAGE

IP and PS are complementary ROM routines that enable the user to switch memory modules on and off without disrupting the calculator. There is no problem switching modules that contain only data, since the 41C doesn't keep track of the SIZE. IP and PS are only needed for switching modules that contain programs.

The calculator continually keeps track of the part of memory currently occupied by programs. The top of program memory is immediately below data register R00. This data/program partition is called the "curtain." The bottom of program memory is defined by the permanent .END.. (The .END. is permanent in the sense that it cannot be deleted. However, it can be moved by inserting or PACKing.)

Pointers to the curtain and to the .END. are maintained in status register c, an important system scratch register (V6N6P20). If the 41C ever finds that the register immediately below the curtain is nonexistent it will give MEMORY LOST. If the .END. pointer is altered, access to CAT 1 will be lost. Therefore, it is essential when switching memory modules containing programs that these two pointers be handled properly. IP and PS store the c register contents, which includes both pointers, in location 256 (the bottom register of the memory module). IP stops there, while PS continues in the new module by recalling register 256 and placing it in status register c.

IP is designed to be used during the page-switching setup procedure. This procedure is best described by the following example.

Example 1: We wish to set up three memory modules for switching. The modules, which can be a mixture of single, dual, and quad density, are placed in a port extender which has a switch for each module. The modules are to be numbered 1, 13, and 45 (the numbering is arbitrary from 0 to 127).

```
Switch on module 1
MASTER CLEAR
XEQ  SIZE as desired*
Load any programs
PACK if desired
XEQ  IP
```

Module 1 is now initialized and ready to be switched off line. Do not resize, pack, or insert program steps after executing IP. You should immediately switch the module off line. If you need to make changes, execute PS first to de-initialize the module. This restriction on resizing does not apply to PS, since PS does not leave anything in register 256 of the active module.

```
Switch off module 1, switch on module 13.
MASTER CLEAR
XEQ SIZE as desired*
Load any programs
PACK if desired
XEQ  IP
Switch off module 13, switch on module 45
MASTER CLEAR
XEQ SIZE as desired*
Load any programs
PACK if desired
Load any key assignments
```

You are now set up for page switching. (See Example 1 of the PS writeup).

TECHNICAL DETAILS

XROM: 10,45	IP	SIZE: 000

Stack Usage:
- 0 T: TEMPORARY c FROM OM
- 1 Z: 240
- 2 Y: Y
- 3 X: X
- 4 L: Y

Alpha Register Usage:
- 5 M: BYTES STORED
- 6 N: CLEARED
- 7 O: CLEARED
- 8 P: CLEARED

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED TO LOWER CURTAIN
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Flag Usage: SEVERAL USED BUT ALL RESTORED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 3

ΣREG: SET TO 000 ABSOLUTE

Data Registers: R00: R06: R07: R08: R09: R10: R11: R12: NONE USED. Absolute location 256 (below the .END.) is used.

Global Labels Called:

Direct	Secondary
PART OF PS	2D
E?	C?
OM	
S?	
VA	
GE	

Local Labels In This Routine:
13
14 TWICE

Execution Time:	About 3 seconds.
Peripherals Required:	A MEMORY MODULE

Interruptible?	YES
Execute Anytime?	NO
Program File:	BL
Bytes In RAM:	60
Registers To Copy:	46

Other Comments: Switch module off line immediately after using IP.

*The SIZE must be low enough so that the .END. is contained in the module, rather than in the basic machine. To check this either XEQ E? and make sure the result is ≥257, or just XEQ IP and see whether you get the OVERSIZE prompt.

COMPLETE INSTRUCTIONS FOR IP

To page switch among N modules, you need to set up the first N-1 of them. This set up procedure consists of the following steps:

1. Turn on the first module (others off)
2. MASTER CLEAR
3. Load any programs and data desired
4. XEQ IP
5. Turn off the module

Repeat steps 1-5 for each of the first N-1 modules. If IP returns a message of OVERSIZE, backarrow and reSIZE to a number less than or equal to the number shown in X. Then resume with step 4. After completing the 5 steps for N-1 modules, perform steps 1-3 for module N. Then load any key assignments you want and you're ready to page switch.

LINE BY LINE ANALYSIS OF IP

Line 71 calls a major portion of the page switching program beginning at line 115. The next few lines check whether the .END. is below register 257. If so, the OVERSIZE error message is generated and the maximum allowable SIZE is computed in lines 133-139. Otherwise the curtain is lowered to 16 (line 126) and the old c register (with its first three nybbles cleared by line 125) is brought into X. The next few lines store the old c register as an alpha constant in register 256, the bottom register of the module. Line 73 restores the old c register.

REFERENCES FOR IP

See *PPC CALCULATOR JOURNAL*, V8N1P25.

CONTRIBUTORS HISTORY FOR IP

Roger Hill (4940) and Keith Jarett (4360) wrote the final ROM version, but Richard Nelson (1), Lee Vogel (4196) and others have made valuable suggestions.

FURTHER ASSISTANCE ON IP

Call Roger Hill (4940) at (618) 656-8825 or during some holiday periods at (213) 794-7376.

Call Richard Nelson (1) at (714) 754-6226.

Routine Listing For:	IP
70◆LBL "IP"	121 R↑
71 XEQ 14	122 SIGN
72 R↑	123 R↑
73 X<> c	124 R↑
74 RDN	125 ΣREG T
75 RTN	126 XROM "OM"
115◆LBL 14	127 STO [
116 XROM "E?"	128 240
117 257	129 ASTO IND X
118 -	130 R↑
119 X<0?	131 R↑
120 GTO 14	132 RTN

NOTES

IR - INSERT RECORD

This routine is called Insert record and can be considered part of a file management system. IR applies to files consisting of fixed length records where each record is a block of consecutive data registers. IR is a special block move routine which makes room between two file records for insertion of a new record. See also the related routine DR.

Example 1: The following list of registers shows an example file consisting of a simplified telephone directory. Use IR to insert a new 5th record in this file. The 5th record is to be the following:

New Record #5:

Paul Jones
223-2654
Albany
NY

This example file consists of a list of names and phone numbers. Only six records are in the file to begin with. Each record consists of 6 consecutive registers with the following format:

1st register holds first name
2nd and 3rd registers hold the last name
4th register holds the telephone number
5th register holds the city name
6th register holds the state name

The records of the original file are assumed to be the following:

Record #1:	Mary Adams 354-1662 Gary, IN
Record #2	Jane Hamilton 363-5648 Boston, MA
Record #3	Robert Jefferson 261-2347 Fresno, CA
Record #4	Mike Johnson 745-3254 Denver, CO
Record #5	James Masterson 565-2314 Toledo, OH
Record #6	Joe Robinson 756-4438 Peoria, IL

This sample file is stored in data registers R10-R45 where each record consists of 6 consecutive data registers.

R10: Mary	R28: Mike
R11: Adams	R29: Johnso
R12:	R30: n
R13: 354.1662	R31: 745.3254
R14: Gary	R32: Denver
R15: IN	R33: CO
R16: Jane	R34: James
R17: Hamilt	R35: Master
R18: on	R36: son
R19: 363.5648	R37: 565.2314
R20: Boston	R38: Toledo
R21: MA	R39: OH
R22: Robert	R40: Joe
R23: Jeffer	R41: Robins
R24: son	R42: on
R25: 261.2347	R43: 756.4438
R26: Fresno	R44: Peoria
R27: CA	R45: IL

Like the other file management routines, IR can expect to find the following information in registers R07, R08, and R09.

R07: starting register of entire file
R08: number of registers per record
R09: total number of records in the file

For the above sample file these numbers are:

R07: 10 = starting register
R08: 6 = number of registers per record
R09: 6 = total number of records

Having stored the data and the file information in the above registers, to insert a new record number 5, simply key in 5 and XEQ "IR". The data registers when the IR routine ends contain the following.

R07: 10	R28: Mike
R08: 6	R29: Johnso
R09: 7	R30: n
R10: Mary	R31: 745-3254
R11: Adams	R32: Denver
R12:	R33: CO
R13: 354.1662	R34: ***
R14: Gary	R35: ***
R15: IN	R36: ***
R16: Jane	R37: ***
R17: Hamilt	R38: ***
R18: on	R39: ***
R19: 363.5648	R40: James
R20: Boston	R41: Master
R21: MA	R42: son
R22: Robert	R43: 565.2314
R23: Jeffer	R44: Toledo
R24: son	R45: OH
R25: 261.2347	R46: Joe
R26: Fresno	R47: Robins
R27: CA	R48: on
	R49: 756.4438
	R50: Peoria
	R51: IL

Note that IR has simply moved the data following record 5 into higher numbered registers to make room in R34-R39 for input of the new record. IR does not acutally insert the new data, IR simply makes room so that the new record may be inserted between previously existing records. Also, IR updated the count of the total number of records in R09. Note that James Masterson is now the 6th record and Joe Robinson is now the 7th record.

COMPLETE INSTRUCTIONS FOR IR

1) A file in the 41C is to consist of a number of fixed length records where each record consists of a consecutive block of registers. Thus the entire file consists of one large block of consecutive registers. As with the other file management routines, IR assumes the following information in registers R07, R08, and R09.

R07: starting register of the entire file
R08: number of consecutive registers per record
R09: total number of records in the file

2) To make room to insert a new kth record, key in k and XEQ "IR".

3) IR will move the records following and including the kth record into higher numbered registers to make room to insert new data for a new kth record. IR will also add 1 to R09 to update the new number of records. Note that this will cause a change in the numbering of the records following and including the old kth record. The IR routine jumps to the block move routine BM.

MORE EXAMPLES OF IR

Example 2: The following matrix was used in Example 1 of the M1 routine. Matrices are assumed to be stored with each row occupying a consecutive block of registers. Thus the number of columns is the block size and the entire matrix is stored row by row as one string of consecutive registers. R07 holds the starting register of the matrix and R08 holds the number of columns. In this manner the storage of matrices corresponds to file storage and vice versa. As a result, the file management routines and the matrix manipulation routines can be used together. In the matrix routines M1 - M5 it is not necessary to store the number of rows in R09, but if either IR or DR is to be applied to a matrix, the user is advised to reserve R09 for the number of rows in the matrix. The following 6x5 matrix is assumed to be stored in registers R15-R44. Use IR to insert a new 3rd row which consists of the following data:

84 97 32 22 54

The original matrix is:

$$\begin{bmatrix} 21 & 35 & 55 & 74 & 83 \\ 11 & 93 & 56 & 36 & 29 \\ 65 & 78 & 32 & 27 & 75 \\ 53 & 94 & 46 & 62 & 97 \\ 54 & 39 & 61 & 67 & 82 \\ 23 & 45 & 77 & 15 & 25 \end{bmatrix}$$

and we show the correspondence between the data registers and the original matrix elements below. The element in the upper left-hand corner is assumed to be in row 1 and column 1. Store the matrix entries in the following registers.

R15: 21	R23: 36	R31: 94	R39: 82
R16: 35	R24: 29	R32: 46	R40: 23
R17: 55	R25: 65	R33: 62	R41: 45
R18: 74	R26: 78	R34: 97	R42: 77
R19: 83	R27: 32	R35: 54	R43: 15
R20: 11	R28: 27	R36: 39	R44: 25
R21: 93	R29: 75	R37: 61	
R22: 56	R30: 53	R38: 67	

Store the following data in R07, R08, and R09.

R07: 15 = starting register of matrix
R08: 5 = number of columns in the matrix
R09: 6 = number of rows in the matrix.

The rows of the matrix correspond to records in a file so to make room to insert a new third row key in 3 and XEQ "IR". Rows 3, 4, 5, and 6 in the original matrix will move down in memory to make room for a new third row. IR does not input the new data but only provides the necessary space. Note also that R09 now contains 7 for the new number of rows. The data registers now contain the following where the *'s are used to indicate where the new row elements are to be stored.

R15: 21	R24: 29	R33: 27	R42: 61
R16: 35	R25: *	R34: 75	R43: 67
R17: 55	R26: *	R35: 53	R44: 82
R18: 74	R27: *	R36: 94	R45: 23
R19: 83	R28: *	R37: 46	R46: 45
R20: 11	R29: *	R38: 62	R47: 77
R21: 93	R30: 65	R39: 97	R48: 15
R22: 56	R31: 78	R40: 54	R49: 25
R23: 36	R32: 32	R41: 39	

Routine Listing For: IR

```
90♦LBL "IR"
91 ISG 09
92 ""
93 XEQ 03
94 ST- T
95 *
96 GTO "BM"

78♦LBL 03
79 RCL 07
80 RCL 08
81 RCL Z
82 *
83 +
84 STO Y
85 RCL 09
86 R↑
87 -
88 RCL 08
89 RTN
```

LINE BY LINE ANALYSIS OF IR

IR uses the BM routine and updates the count in R09. The input required of BM is given in the following stack configuration where s=starting register of the file, c=number of registers per record, n=number of records in the file. i= user input to IR.

Z: first reg. = s + c*(i-1)
Y: destination of first register = s + c*i
X: number of registers = c*(n-i+1)

Lines 90-95 set up these values in the stack before control is transferred to BM.

CONTRIBUTORS HISTORY FOR IR

The IR routine and documentation were written by John Kennedy (918).

FINAL REMARKS FOR IR

IR is barely the start of a file management system.

FURTHER ASSISTANCE ON IR

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

NOTES

TECHNICAL DETAILS

XROM: 20, 37	IR	SIZE: depends on file size

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊦: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
- R00: not used
- R06: not used
- R07: s=start reg. of file
- R08: c=# reg. per record
- R09: n=# records in file
- R10: not used
- R11: not used
- R12: not used

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine: 03

Execution Time: depends on file size and configuration as well as the number of the record inserted.

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: M2

Bytes In RAM: 36

Registers To Copy: 61

Other Comments:

No special SIZE requirement is necessary provided the data block(s) already exist

APPENDIX G CONTINUED FROM PAGE 227.

often called microcode. Each HP-41C function requires many such instructions. Microcode and Assembly Language are often used interchangeably. In the strict sense the latter more nearly describes what is used by the HP-41 microprocessor.

N

N REGISTER - The seventh register (006 absolute) of HP-41 memory--the second seven byte group from right end of the ALPHA Register. N is one of 16 status registers (R00 through R15 absolute) recorded during a WSTS operation. N may be used for general scratch and non-normalized STO/RCL. Also see ALPHA Register.

NATURAL NOTATION - The representation of HEX characters 3A thru 3F as they display. See NH routine. The correspondence is shown below. Natural Notation is usually faster to use in programs such as **NH**.

HEX	'Natural'	HEX	'Natural'
3A	:	3D	=
3B	,	3E	>
3C	<	3F	?

NIBBLE - See Nybble. This is probably a preferred spelling. Some users spell Nibble with a "y".

NNN - Non-Normalized-Number, an early HP term describing an undefined number in an HP calculator register. If a nybble other than a 0 or 9 appears in either of the sign 'digits' the number is classified as an NNN. In the HP-41 the first Nybble is 0 for positive number, 9 for negative number, and 1 for Alpha data. All others are NNN's. Also see PPC Member Handbook, 2nd Ed., page 150 for historical references.

NOP - No Operation. An instruction not found on the HP-41C, but common to most computers and some early HP calculators. It is a "do-nothing" space filler that may be used with ISG and DSE to make a simple count-by-one counter. A Text zero, HEX F0, decimal 240 byte is a good HP-41 NOP.

NORMALIZATION - A process performed by the HP-41C on non-ALPHA data when recalled from a non-status register. If you are working with synthetics, normalization can be a major frustration; if you are working with numerical calculations, normalization is absolutely essential to preserve accuracy.

NULL - An instruction (HEX 00) used by the HP-41 operating system as:

a. A filler for a deleted instuction.
b. A separator for sequential numbers in memory.
c. A filler for bytes that have no value, such as the assigned key in a LBL instruction.

Null instructions as used in "a." are removed during a PACKING operation.

NYBBLE (NIBBLE) - Four BITS or one half a byte. The HP-41 CLD instruction is decimal 127 and 0111 1111 in Binary. The two nybbles 0111, 1111 may be represented in HEX as 7F.

O

O Register - The eighth register (007 absolute) of HP-41 memory--the third seven byte group from right end of the ALPHA Register. O is one of 16 status registers (R00 thru R15 absolute) recorded during a WSTS operation. O may be used for general scratch and non-normalized STO/RCL. Also see ALPHA Register.

P

P REGISTER - The ninth register (008 absolute) of HP-41 memory--the left seven bytes at the ALPHA Register. P is one of 16 status registers (R00 through R15 absolute) recorded during a WSTS operation. P must be used with care for general scratch and non-normalized STO/RCL, because the left most 4 bytes are used by the micro-processor as scratch during program execution of AVIEW, VIEW, and number entry program lines.

PACKED END - An end instruction that has a specific BIT set that tells the HP-41 operating system that no editing has taken place since the last PACK. This saves PACKING TIME, because the packed end says "skip this program file" it is already PACKED". Also see program and program file.

PAGE SWITCHING - A memory expansion concept proposed by Richard Nelson (1) (see PPC CJ, V8N1P25c) as a means of switching QUAD modules on and off the 41 bus. The page switching concept was implemented by Roger Hill (4940) and Keith Jarett (4360) in the PPC ROM with the **IP** and **PS** routines.

PORT EXTENDER - A 41C accessory that plug into one of your ports and includes several ports. The port extender allows you to run more than four accessories. HP does not manufacture a port extender as of November 1981.

POSTFIX - An HP term used to describe the second and subsequent bytes of a multibyte instruction of the HP-41 HEX table. The first half of the table (rows 0 thru 7) are postfix direct and the second half of the table (rows 8 thru F) are postfix indirect instructions. Also see Prefix, and PPC J, V6N5P23 HEX table.

PREFIX - A term used by Hewlett-Packard in the CORVALLIS DIVISION COLUMN in PPC J, V6N5P20b to describe the HP-41 HEX table instruction organization. Two byte instructions are composed of a prefix byte and a post fix byte. Prefix bytes come from rows 9 thru F of the HEX table.

PREFIX MASKER - One of a class of key assignments (for example 247,63) that masks bytes of following instructions when inserted in a program. All key assignments from F3 through FF are prefix maskers. They also byte jump in RUN mode. See PPC CJ, V8N1P31 and V8N5P11.

PROGRAM - A sequence of instructions that perform a planned task. A program may or may not have a label--Local or Global. The HP-41 may have many programs in memory. Also see Program File.

PROGRAM FILE - The sequence of instructions between:

a. "Top" of program memory and an END
b. Between two END's
c. Between an END and the .END.

The HP-41 instructions COPY and PRP operate only on program files and not specific Global labels as many HP-41 users expect.

APPENDIX G CONTINUED ON PAGE 251.

JC - JULIAN DAY NUMBER TO CALENDAR DATE

This calendar routine will convert a Julian Day Number (JDN) to a calendar date. The date returned may be interpreted for either the Gregorian or Julian calendar depending on a flag setting. The valid range is from March, year 0 A.D. The output is of the form with the year in Z, the month in Y, and the day number of the month in X. See also the routine CJ. This routine is the inverse of CJ.

Example 1: Compute the Gregorian calendar date of the Julian Day Number 2,444,790.

Clear flag 10 for the Gregorian calendar. Key in 2,444,790 and XEQ "JC". The routine returns with the following values in the stack.

Z: year = 1981
Y: month = 7
X: day = 4

The date is July 4, 1981.

Example 2: Find the date which is 93 days after July 4, 1981.

From Example 1 we know the JDN of July 4, 1981 is 2,444,790. Adding 93 to this number gives 2,444,883 as the Julian Day Number of the unknown date. Assuming flag 10 is still clear from the previous example, key in 2,444,883 and XEQ "JC". The stack returns with:

Z: 1981
Y: 10
X: 5

The desired date is October 5, 1981.

BACKGROUND FOR JC

See the CJ routine for background on Julian Day Numbers (JDN's), the Gregorian and Julian Calendars, and the conventions used here. As with CJ, when a JDN is converted to a calendar date, only the integer part of the input is considered, and the resulting output is the calendar date at noon of which the JDN begins to apply.

COMPLETE INSTRUCTIONS FOR JC

1) Clear flag 10 for Gregorian calendar dates and set flag 10 for Julian calendar dates.

2) Input the Julian Day Number in X and XEQ "JC". The calendar date is returned in the stack as:

Z: year (0-)
Y: month (1-12)
X: day of month (1-31)

The year is also duplicated in T (not shown above).

An improper flag setting or date output will cause "DATA ERROR".

MORE EXAMPLES OF JC

Example 3: Compute the Gregorian calendar date for the JDN = 2,361,221.

Clear flag 10. Key in 2,361,221 and XEQ "JC". The result returned in the stack is:

Z: 1752
Y: 9
X: 13

The Gregorian calendar date is September 13, 1752.

Example 4: Compute the date under the Julian calendar for the JDN of Example 3 above.

Set flag 10. Key in 2,361,221 and XEQ "JC". The result returned in the stack is:

Z: 1752
Y: 9
X: 2

The Julian calendar date is September 2, 1752.

Example 5: Find the Gregorian calendar date of the JDN = 2,451,545.

Clear flag 10. Key in 2,451,545 and XEQ "JC". The stack returns:

Z: 2000
Y: 1
X: 1

The date is January 1, 2000.

Example 6: Find the date which preceeds August 14, 1981 by 159 days.

Clear flag 10. Use CJ to compute the JDN for August 14, 1981. (Key 1981 ENTER↑ 8 ENTER↑ 14 and XEQ "CJ"). This results in 2,444,831. Now subtract 159 from this number and XEQ "JC". 159 - XEQ "JC".
The stack returns:

Z: 1981
Y: 3
X: 8

The date is March 8, 1981.

FURTHER DISCUSSION OF JC

Validity Range for JC

Routine JC may be used on any JDN as long as the output date does not fall before March 1, 0 A.D. (=1 B.C.) of the calendar under consideration. A DATA ERROR message will appear if this condition is violated, to remind the user that the routine is not valid for B.C. years. (See also the application routine which uses JC and is valid for BC years). It is up to the user to decide whether he/she wishes to use the Gregorian or Julian Calendar; using the routine with the "wrong" calendar for the date being converted will still give the correct extrapolation of the calendar to that date. (The changeover from the Julian to Gregorian Calendar in Rome, incidentally, occurred between JDN 2,299,160 and 2,299,161.)

The maximum JDN for which JC is valid is limited only by the 10-digit precision of the calculator. The routine seems to work for JDN's at least 1,000,000 years in the future, but between then and 10,000,000 years in the future round-off errors begin to occur during the calculations. As with CJ, this routine is valid for dates far beyond those for which our present calendars are likely to be used.

This routine, like CJ, does not include the proposed calendar correction which would make 4000 and its multiples non-leap years (see Background on CJ).

APPLICATION PROGRAM 1 FOR JC

The following sequence of instructions will convert a Julian Day Number in X to the Gregorian Calendar date in the format YYYY.MMDD in the X-register:

```
CF 10
XROM JC
E2          (see note following
/           the first application
+           routine using CJ for
E2          the creation of the
/           E2 instruction)
+
FIX 4       14 bytes
```

APPLICATION PROGRAM 2 FOR JC

The follwing sequence will convert a JDN in X to the Gregorian Calendar date in X with the format MM.DDYYYY:

```
CF 10
XROM JC
E2
/
+
X<>Y
E6
/
+
FIX 6            15 bytes
```

APPLICATION PROGRAM 3 FOR JC

The following sequence will, given the JDN in X, display the Gregorian Calendar date in the alpha format MM/DD/YYYY

```
CF 10
XROM JC
FIX 0
CF 29
CLA
ARCL Y
"├ /"
ARCL X
"├ /"
ARCL Z
AVIEW               22 bytes
```

APPLICATION PROGRAM 4 FOR JC

The following sequence will convert a JDN in X to the Gregorian Calendar date in X with the format DDMM.YYYY

```
CF 10
XROM JC
E4
ST / T
SQRT
*
+
+
FIX 4
CF 29               16 bytes
```

APPLICATION PROGRAM 5 FOR JC

The following sequence will, given a Julian Date (including the fractional part) in X, produce the date in Y in the format YYYY.MMDD and the time in X in the format HH.MMSS: (Register R00 may be of course replaced by any other unused register. See also the warning on accuracy follwing Application Program 6 of the CJ routine).

```
.5          E2
+           /
ENTER↑      +
FRC         E2
24          /
*           +
HMS         RCL 00
STO 00      FIX 4
X<>Y
XROM JC                24 bytes
```

Here the calendar to be used depends on the user's setting of Flag 10. Inserting the following steps before XROM JC, however, will cause the correct calendar to be chosen automatically (assuming the 1582 adoption of the Gregorian Calendar as described in the background for CJ.

```
2299161
X<>Y
CF 10
X<Y?
SF 10          (13 additional bytes)
```

Also, the instructions between XROM JC and RCL 00 may be modified along the lines of routines (2), (3), or (4) to accomodate other date formats.

APPLICATION PROGRAM 6 FOR JC

The program JCA is used in exactly the same manner as JC but with the following additional features. For Julian Days between -1785282 (3/1/9601 BC) and 1721788 (12/31/1 AD), JCA adds 3506400 to the Julian Day Number and subtracts 9601 from the corresponding year. BC years are displayed as negative numbers. For astronomical sequencing of BC years (2AD, 1AD, 0AD, 1BC, 2BC), set Flag 4. If Flag 4 is clear the sequencing of BC years is assumed to be 2AD, 1AD, 1BC, 2BC. JCA does not change the status of Flag 4 once set by the user. Flag 10 will be automatically set for Julian Days prior to 2299161 (10/15/1582) and will be cleared after execution. Flag 10 must still be set manually for Julian Calendar dates after 2299160. To facilitate chaining of operations, program JCA ensures that the original contents of the X-register prior to entering the Julian Day Number will end up in the T-register. If DATA ERROR is displayed, the Julian Day Number entered was prior to -1785282.

APPLICATION PROGRAM FOR: JC

```
01♦LBL "JCA"        18 XROM "JC"
02 CF 09            19 RCL Z
03 2299161          20 9601
04 X>Y?             21 FC?C 09
05 SF 10            22 CLX
06 RDN              23 -
07 1721789          24 X<0?
08 X>Y?             25 FC? 04
09 SF 09            26 X=0?
10 RDN              27 ISG X
11 3506400          28 ""
12 FS? 09           29 STO T
13 ST+ Y            30 X<> [
14♦LBL 01           31 RDN
15 RCL Z            32 CF 10
16 STO [            33 .END.
17 RCL Z
```

FORMULAS USED IN JC

Let N be the number of days that have elapsed since the beginning of March of 0 A.D. Thus,

(1) N = JDN - 1,721,119

The average length of a Gregorian century is 36524.25 days, and the number of whole Gregorian centuries that have elapsed since March of 0 A.D. is:

(2) C = INT((N - e)/365.2425)

where e is any number between 0 exclusive and .25 inclusive. (In JC we use e = .2) The number

(3) N' = N + C - INT(C/4)

then gives the number of days since March, 0 A.D., that would have elapsed if all century years were leap years (as they acutally were in the Julian Calendar), with a year being 365.25 days on the average. If the Julian Calendar has been selected, equation (3) is replaced by

(3') N' = N + 2

this being possible due to the fact that both calendars agree during the time that C=2. From here on the calculations are the same for both calendars.

Let Y' and M' be the "shifted" year and month, where a "shifted year" runs from March (M'=0) of the same numbered ordinary year to February (M'=11) of the next ordinary year. Then Y' (which can also be thought of as the number of whole shifted years that have elapsed since the beginning of March 0 A.D.) is given by

(4) Y' = INT((N' - e')/365.25)

where e' is any number satisfying the same restriction as e (again we have used e'=.2 in the actual program). The number of days that have elapsed since the beginning of the shifted year is

(5) N'' = N' - INT(365.25*Y')

and the shifted month can be found by

(6) M' = INT(((N'' - d)/30.6)

where d is any number between .4 and .6 (exclusive); we have used d=.5 in JC . (The number 30.6 is the average length of a month during each 5-month cycle, beginning with March and lasting until the end of February.) The day of the month is then

(7) D = INT(N'' - 30.6*M' + d')

where d' has the same restrictions as d (and has again been chosen to be .5 in our program). Finally, the ordinary year and month Y and M can be obtained from the shifted year and month by

(8) Y = Y' and M = M' + 3 if M'<=9

Y = Y' + 1 and M = M' - 9 if M'>9

Routine Listing For: JC

```
157♦LBL e           182 INT
158♦LBL "JC"        183 ST* Y
159 INT             184 RDN
160 1721119.2       185 INT
161 -               186 -
162 ENTER↑          187 .3
163 FS? 10          188 -
164 -2              189 STO Y
165 FS? 10          190 30.6
166 GTO 09          191 ST/ Y
167 36524.25        192 X<>Y
168 /               193 INT
169 INT             194 *
170 ST+ Y           195 ST- Y
171 4               196 ISG Y
172 /               197 X<> L
173 INT             198 -3
174♦LBL 09          199 X↑2
175 -               200 X<Y?
176 X<0?            201 ISG T
177 SQRT            202 X<> L
178 STO Y           203 -
179 365.25          204 X<>Y
180 ST/ Y           205 INT
181 X<>Y            206 END
```

LINE BY LINE ANALYSIS OF JC

JC begins by truncating the input to an integer and subtracting 1,721,119.2 (lines 159-161) to give the quantity N - .2 in the X-register. If the Gregorian Calendar has been selected (Flag 10 clear) then C and N' - .2 are found using equations (2) and (3) (lines 167-175); if the Julian Calendar has been selected (Flag 10 set) then N' is found using equation (3') (lines 163-166 and 174-175). At this point the quantity N' - .2 is checked for positiveness (lines 176-177); a negative value would indicate a date earlier than March 1, 0 A.D. of the calendar being used. Equation (4) is implemented in lines 178-182 to get Y', and after using equation (5) and subtracting an additional .3 in lines 183-188 we have N'' - .5 in the X-register while retaining Y' in the Z- and T-registers. Lines 189-193 use equation (6) to get M', and after line 197 we have in the stack:

T: Y'
Z: Y'
Y: N'' - 30.6*M' + .5
X: M'

Because Y always has a fractional part of .1, .3, .5, .7, .9, it always gets increased by 1 by the ISG Y of line 196 with no skipping since the integer part never exceeds 31. Lines 198-203 use some more ISG trickery to convert Y' and M' to Y and M. After line 199 we have in the stack:

T: Y'
Z: N'' - 30.6*M' + .5
Y: M'
X: 9

L: -3

If M'>9 then Y' is increased by 1 in line 201, and line 202 is skipped so that 9 is subtracted from M' in line 203. But if M'<=9 then line 201 is skipped, and the 9 and -3 get exchanged in line 202 so that 3 gets added to M' in line 203. Lines 204-205 complete the implementation of equation (7) to give D, with all three output variables Y, M, D ending up in their proper places. The year, incidentally, is also duplicated in the T-register.

REFERENCES FOR JC

1. "Gregorian Calendar" 65 NOTES, HP-25 Library V4N5P20,P26

2. Dan M. Fenstermacher, "Calendar Algorithms", PPC JOURNAL V5N1P16

3. "CALENDARS", User's Library Solutions, For The HP-67/97, The Hewlett-Packard Company

4. "CALENDARS", User's Library Solutions, For The HP-41C, The Hewlett-Packard Company

5. "Calendar", Encyclopedia Brittanica (Contains much detailed information on various calendar systems, the calculation of Easter, etc.)

6. Gordon Moyer, "The Origin of the Julian Day System", SKY AND TELESCOPE, V61 N4 P311 (April 1981) Also contains algorithms for converting to and from Julian Day Number. (Corrections in June 1981, p. 550, and in July 1981, letter to the editor, p. 16)

7. THE ASTRONOMICAL EPHEMERIS (previously, THE AMERICAN EPHEMERIS AND NAUTICAL ALMANAC), U.S. Government Printing Office, Washington, D.C. (any year)

8. Explanatory Supplement to THE ASTRONOMICAL EPHEMERIS and THE AMERICAN EPHEMERIS AND NAUTICAL ALMANAC, 1961, p.414 (and other editions)

9. C.W. Allen, ASTROPHYSICAL QUANTITIES, Athlone Press, London, 1976 (Examples of JDN on p. 295)

CONTRIBUTORS HISTORY FOR JC

The JC routine and documentation are by Roger Hill (4940). David Spear (5488) provided some additional information on calendars and is the author of the application program JCA.

FINAL REMARKS FOR JC

JC is optimized for the HP-41C calculator but the same formulas may be implemented on any machine.

FURTHER ASSISTANCE ON JC

Roger Hill (4940) phone: (618) 656-8825
Fernando Lopez-Lopez (2287) phone: (714) 421-9791 after 9PM

TECHNICAL DETAILS

XROM: 20, 22 — **JC** — SIZE: 000 minimum

Stack Usage:	Flag Usage:
0 T: used	04: not used
1 Z: used	05: not used
2 Y: used	06: not used
3 X: used	07: not used
4 L: used	08: not used
Alpha Register Usage:	09: not used
5 M: not used	10: clear=Gregorian set=Julian
6 N: not used	
7 O: not used	
8 P: not used	25: not used
Other Status Registers:	**Display Mode:**
9 Q: not used	
10 ⊢: not used	not used
11 a: not used	**Angular Mode:**
12 b: not used	not used
13 c: not used	
14 d: not used	**Unused Subroutine Levels:**
15 e: not used	5
ΣREG: not used	**Global Labels Called:**
Data Registers:	Direct: none — Secondary: none
R00: not used	
R06: not used	
R07: not used	
R08: not used	
R09: not used	
R10: not used	
R11: not used	**Local Labels In This Routine:**
R12: not used	e, 09

Execution Time: 3.1 seconds if F10 is clear; 2.6 seconds if F10 is set

Peripherals Required: none

Interruptible? yes	Other Comments:
Execute Anytime? no	
Program File: BD	
Bytes In RAM: 98	
Registers To Copy: 53	

L- - LOAD PART OF LB

L- and -B together comprise a subroutine version of LB. L- initializes the byte loading process without any prompting, returning to the calling program. Bytes can be loaded one by one by placing the decimal code in X and executing -B.

Example 1: The following program segment prompts for input and loads an XROM instruction into program memory (after the user has supplied the usual LBL "++" ++...+ XROM LB sequence). This program checks for sufficient SIZE, converts the XROM numbers Y and X to decimal codes for LB, and loads two bytes from the decimal codes. It then prompts for another pair of XROM numbers.

```
01 LBL "XLB"
02 12
03 XROM VS
04 FC?C 25
05 PROMPT
06 XROM L-
07 CF 22
08 LBL 00
09 "XROM Y, X  ?"
10 PROMPT
11 FC?C 22
12 GTO 01
13 XROM XL
14 X<>Y
15 STO 05
16 X<>Y
17 XROM -B
18 RCL 05
19 XROM -B
20 GTO 00
21 LBL 01
22 CF 09
23 XROM -B
24 END
```

To use "XLB" first key in the LBL "++" ++...+ XROM LB sequence as described in the instructions for LB. Then XEQ "XLB" and supply two XROM numbers in response to the prompt. For instance for XROM 10, 00 key in 10 ENTER↑ 0. Press R/S to calculate and load two bytes. When the next prompt for XROM numbers appears you can either enter another pair of numbers or press R/S without an input to terminate the byte-loading process. The usual prompt "SST, DEL 00p" will be given.

Example 2: If you change line 23 of "XLB" (Example 1) from CF 09 to CF 08, pressing R/S without an input will not terminate the byte loading. Instead, the CF 08 instruction switches to the manual LB operation, allowing additional bytes to be loaded from the keyboard.

COMPLETE INSTRUCTIONS FOR L-

These routines allow bytes to be loaded under the control of your own program. The general rules for their use are as follows:

1. In the program that you are writing which controls the loading of bytes, put the instruction XROM L-. This initializes the byte-loading process and then (instead of prompting for Byte #1) returns to your control program.

2. Have your control program calculate or otherwise place each byte (only decimal allowed in this version) in the X-register, and put an XROM -B in your program to load that byte. Flags 22 and 23 are ignored, as well as whether the calculator is in ALPHA or non-ALPHA mode. The call to routine -B causes one byte to be loaded and then returns to your control program.

3. To terminate the byte-loading process, put the instructions CF 09, XROM -B in your control program. Executing the routine -B with Flag 09 cleared will cause no additional byte to be loaded, but rather a termination of the byte loading, in this case *not* returning to your control program, but ending with a "SST, DEL 00*p*" prompt. Executing -B with Flag 08 cleared will switch from automatic to manual byte loading, allowing more bytes to be loaded directly from the keyboard.

4. Before running your control program, check for size 12, and make room in program memory where you want the bytes to be loaded in exactly the same way as when using the prompting version LB. That is, key in (in PRGM mode) LBL"++", a string of +'s, and XROM LB.

5. Switch out of PRGM mode and instead of pushing R/S to start the byte loader, execute your own control program. Then sit back while your program (if correctly written) calculates, prompts for, or otherwise creates and loads each byte.

6. Execution will terminate with the "SST, DEL 00*p*" prompt, whereupon you can perform the "cleanup" operations just as with the ordinary LB program.

7. If you want your control program to correct a byte that it previously loaded, have it enter a negative number in X and execute -B to get rid of the last-entered byte.

8. Your control program is welcome to make use of any of the contents of registers 06-11 (see above), as long as it doesn't change any of these registers.

WARNING: Don't execute -B (or let your program do it) without having first initialized the process by executing L- ! A few flag and other safeguards have been incorporated, but executing -B by itself *could* cause MEMORY LOST or destruction of existing programs.

When used properly, L- and -B can be very powerful, ultimately allowing one to write a program which writes programs! A somewhat less exotic application is a byte loading program which allows bytes to be scanned in by a wand instead of keyed in.

APPLICATION PROGRAM 1 FOR L-

The following program "LBW" (Load Bytes With Wand) allows bytes to be loaded by scanning 2-byte paper keyboard (type 5) barcodes. Only the second byte of the barcode is loaded into program memory, but in order to avoid scanning errors the entire barcode is checked for checksum consistency. Using this program along with a barcode hex table (such as in PPC CJ, V7N6P25-26) and HP's Wand Paper Keyboard, one can rapidly scan in the bytes to be loaded in a manner which for many functions is similar to the normal use of the paper keyboard.

For example, the synthetic instruction X<> can be obtained by scanning X<> in the Paper Keyboard (which will supply the correct prefix) and then byte 78 (hex) in the hex table for the postfix, and TONE 26 can be obtained by scanning TONE from the paper keyboard and byte 26 (decimal) from the hex table. For alpha characters, the barcode hex table can be used, but not the alpha character codes in the Paper Keyboard (or in the character table of PPC CJ, V7N6P23) which use a different format for encoding the character.

BAR CODE ON PAGE 483

APPLICATION PROGRAM FOR: **L —**

```
01♦LBL "LBW"
02 XROM "L-"
03♦LBL 01
04 FIX 0
05 CF 29
06 "W: "
07 ARCL 06
08 "⊢ OF "
09 ARCL 07
10 XROM "VA"
11 TONE 7
12 .
13 SF 22
14 XROM 27,05
15 FS? 22
16 GTO 11
17 2
18 X≠Y?
19 GTO 14
20 RCL 01
21 16
22 XROM "QR"
23 RCL 02
24 +
25 X≠0?
26 15
27 X≠0?
28 MOD
29 X=0?
30 X<> L
31 X≠Y?
32 GTO 12
33 RCL 02
34♦LBL 11
35 XROM "-B"
36 GTO 01
37♦LBL 14
38 SIGN
39 X≠Y?
40 GTO 14
41 90
42 RCL 01
43 X=Y?
44 GTO 10
45 189
46 X≠Y?
47 GTO 12
48♦LBL 03
49 -1
50 GTO 11
51♦LBL 14
52 X<>Y
53 X=0?
54 GTO 10
55 5
56 X<Y?
57 GTO 13
58♦LBL 12
59 "BAR/CKSM ERR"
60 XROM "VA"
61 TONE 1
62 GTO 01
63♦LBL 10
64 CF 09
65 GTO 11
66♦LBL 13
67 "LOAD ABORT"
68 TONE T
69 PROMPT
70 GTO 13
71 .END.
```

NOTE: This program also appears under **B**.

Instructions for using "LBW" are as follows:

1. Insert LBL ++, a string of +'s, and XROM **LB** in the desired part of program memory, just as when using **LB**.
2. Switch out of program mode and XEQ "LBW". (Note: SIZE 012 or greater is required. If you get the insufficient SIZE message, re-size the calculator and then key in XEQ "LBW" again to restart the process. Just pushing R/S after re-sizing will cause the ordinary **LB** byte loading version to be initiated instead of the wand version.
3. At each prompt "W: N OF M", scan in the appropriate 2-byte barcode (the WNDSCN command is in effect here). After verifying the checksum, the second byte of the barcode will be loaded.
4. A decimal entry can be made directly from the keyboard by clearing the "W: N OF M" prompt (using ←), making the entry, and pushing R/S. Flag 22 is used to detect such an entry. Afer loading the byte, the program will resume with the "W: N+1 OF M" prompt. Hexadecimal entries are not provided for in this program however.
5. To correct an entry, either (a) scan the 1-byte ←barcode, or (b) clear the prompt and XEQ 03, or (c) clear the prompt, enter a negative number, and push R/S. Method (a) can be used to conveniently clear up to 3 bytes by making up to 3 scans at once and waiting while they are processed one by one.
6. During the prompt for a new byte, X=0 while Y= decimal value of previous byte. If you wish to clear the prompt to check the previous byte value, make elementary calculations, etc., push XEQ 01 afterward to get a re-prompt before continuing with the loading.
7. To terminate the byte-loading process, either (a) scan the one-byte . (decimal point) barcode, or (b) push R/S twice. Then follow the usual "clean-up" procedures as with **LB**. The loading process will also terminate itself automatically after the maximum number of bytes is reached.
8. If you have accidentally terminated and wish to add more bytes or make corrections, push GTO 03 R/S or GTO 01 R/S (rather than XEQ 03 or XEQ 01, which would disable the return to the "LBW" program).
9. Scanning any 1-byte barcode other than ← or . or any barcode of 3 to 5 bytes will cause the message "BAR/CHKSM ERR" and a re-prompt. The same applies to a 2-byte barcode whose checksum does not check. However, scanning a 6-byte or longer barcode will cause vital information in R06-R11 to be wiped out, so in such a case the whole process is terminated with a "LOAD ABORT" message.

To give a brief analysis of the program:

Lines 01-23 initialize the loading process, and lines 03-14 set up the prompt and execute the WNDSCN command. Lines 15-16 detect an entry from the keyboard and branch to lines 34-35 to load the byte (or backup, if the entry is negative). Otherwise a scan with the wand is assumed to have occured, in which case WNDSCN causes the number of bytes to be in X and the decimal byte values in R01-R0k. If k≠2, a branch is made (lines 17-18) to line 37; otherwise the 4-bit wraparound checksum of the last 3 nybbles is calculated and compared with the first nybble (lines 20-32). A mismatch causes a branch at line 32 to LBL 12 (line 58) where the error message is given; otherwise the second byte of the barcode is recalled and loaded (lines 33-35) and we start over (line 36). If k≠2 then we had branched to line 37, after which we check for k=1, and if true we check whether the one byte was 90 decimal (5AH, the decimal print code) or 189 (BDH, the back-arrow code) and branch accordingly, otherwise branching to the error message. Lines 51-57 deal with the case where k is neither 1 nor 2; if k=0 then no scan has taken place and it is assumed that R/S R/S was pushed, so we branch to line 63 and initiate the termination procedure by clearing flag 09. If k>5 we branch to line 66 to produce the "LOAD ABORT" message. For other values of k the "BAR/CHKSM ERR" message is produced in lines 58-61, and line 62 branches to a re-prompt.

*The checksum can be calculated by adding up the decimal values of the nybbles; if the result is zero proceed no further. Otherwise take the result mod 15 and if the result of that is zero, change it to 15. In the present case, we are concerned with the last nybble (call it n2) of R01 and both nybbles (call them n3 and n4) of R02, and since n2 + n3 + n4 is equivalent to n2 + 16*n3 + n4 when taken mod 15, it is necessary to decompose the byte in R02 (=16*n3 + n4) into its separate nybbles before adding. Routine

QR is used, however, to decompose the byte in R01 into its separate nybbles n1 and n2; n1 is the number to be compared with the calculated checksum.

APPLICATION PROGRAM 2 FOR L-

As an example of a program which writes programs, the following program, "COMP", composes random music by generating a program consisting of tone instructions selected at random from tones 0 through 127 using routine RN to generate the random numbers. To use it, initialize the desired section of program memory with the usual LBL ++, string of +'s, and XROM LB, and then go into non-PRGM mode, make sure the SIZE is at least 012, and execute "COMP". The program will prompt for a seed; enter any number and push R/S. The tone instructions will be loaded into program memory until there is no room left, whereupon the usual "SST, DEL OOP" termination will occur. After performing the usual cleanup operations you can execute your newly composed program and hear the music.

This program can be directly compared with Application Program 2 for TN, "MUS", which generates and plays the tones in "real time". The generation of the random numbers is exactly the same for the two programs (see the description of "MUS" under TN for an explanation), and the tones produced by "MUS" and "COMP" for a given initial seed, will be the same up to the point where the latter runs out of memory space. "MUS" has the advantage of producing tones indefinitely with no initial compilation time, but the listener must put up with the approximately 2-second delay between tones, making the "music" rather tedious. "COMP" requires an initial compilation time (3-4 minutes to generate a 49 tone sequence) and the length of the piece is limited by the number of +'s initially put into program memory, but once the compilation is done the music can be played with no intertone delays. Thus, the results of "COMP" (though they may not become instant hits) are likely to be much more satisfying to the listener.

Lines 02-13 of "COMP" initialize the random numbers (see "MUS" under TN), store frequently used constants, and initialize the byte-loading procedure. Lines 14-21 take the integer part of R07, which is the maximum number of bytes that can be loaded, determine whether it is even or odd, and load a null byte (line 21) if the number is odd. This ensures that there is an even number of bytes left over that can be loaded so we can simply load tone instructions repeatedly (at 2 bytes per instruction) until we run out of bytes, at which time -B will terminate the loading automatically, and the termination will not be in the middle of an instruction. Lines 22-30 form the tone-loading loop in which we first (lines 23-24) load byte 159 (decimal) corresponding to the TONE prefix, then obtain a random number whose integer part is uniformly distributed from 0 to 127 (lines 25-28) and use it for the postfix byte (line 29).

As an aid to the mass production of music (or other byte loading operations) one can record on a single track of a card the following 112-byte program: LBL ++, a string of 104 +'s, XROM LB. (When recording and reading this card there will be a prompt for side 2 which you can ignore and clear). Reading this card and using any of the versions of the byte loader will always allow exactly 98 bytes to be loaded, on our present case allowing 49 tones. The final 49-note piece will then fit onto one track of a card with a few bytes to spare for labels, etc.

BAR CODE ON PAGE 483

APPLICATION PROGRAM FOR: L-

```
01♦LBL "COMP"      16 2
02 "SEED?"         17 MOD
03 PROMPT          18 1
04 ABS             19 -
05 LN              20 X=0?
06 ABS             21 XROM "-B"
07 FRC             22♦LBL 01
08 STO 00          23 RCL 04
09 159             24 XROM "-B"
10 STO 04          25 CLX
11 128             26 XROM "RN"
12 STO 05          27 RCL 05
13 XROM "L-"       28 *
14 RCL 07          29 XROM "-B"
15 INT             30 GTO 01
                   31 .END.
```

As an example of HP-41 generated music, the author found particularly nice the 49-note piece (which coincidentally, takes just 49 seconds to play) obtained by using the card described in the last paragraph and inputting a seed of 4; the initial compilation took 3.25 minutes. If however, the user is not so enthralled by this particular composition, he has plenty of others to choose from. And whether or not he would agree that such music is a manifestation of the true soul of the HP-41, it is undeniable that all of this is an interesting example of calculator composed music, programs that generated programs, and the art of synthetic programming in general. Further refinements could include, for example, weighting factors to favor (say) the short duration tones, and even some "rules of composition" to produce particular musical effects.

LINE BY LINE ANALYSIS OF L-

See LB.

CONTRIBUTORS HISTORY FOR L-

L- and -B were conceived and written by Roger Hill (4940) as an integral part of the ROM version of LB

FURTHER ASSISTANCE ON L-

Call William Cheeseman (4381) at (617) 235-8863.
Call Roger Hill (4940) at (618) 656-8825.

Routine Listing For: L-

```
01♦LBL 00              20 STO \        39 X<> [
02 STOP                21 SF 08        40 STO a
03 GTO "++"                            41 X<> \
                       22♦LBL 13       42 X<> b
04♦LBL "LB"            23 12
05 FS? 50              24 XROM "VS"    43♦LBL 14
06 GTO 00              25 FC?C 25      44 XROM "RT"
07 "DEC/HEX INPT"      26 PROMPT       45 117
08 XROM "VA"           27 CLST         46 X<>Y
09 CF 08               28 STO 06       47 -
10 GTO 13              29 SIGN         48 7
                       30 ENTER↑       49 XROM "QR"
11♦LBL "L-"            31 ENTER↑       50 ST- Z
12 CLA                 32 R↑           51 X<>Y
13 XROM "VA"           33 GTO "++"     52 CHS
14 CF 08                               53 STO 09
15 RCL a               34♦LBL 00       54 X<> Z
16 STO [               35 RCL b        55 LASTX
17 RCL b               36 FC? 08       56 XROM "QR"
18 FS? 08              37 GTO 14       57 1.001
19 GTO 14              38 CLD
```

Routine Listing For:	L —

```
 58 ST* 09
 59 ST+ Y
 60 FRC
 61 ST* T
 62 X<>Y
 63 R↑
 64 +
 65 *
 66 X<>Y
 67 X<=0?
 68 GTO 10
 69 ST+ 09
 70 7
 71 *
 72 +
 73 STO 07
 74 XROM "OM"
 75 X<> c
 76 STO 10
 77 CLST

 78♦LBL 06
 79 STO 11
 80 CLA

 81♦LBL 07
 82 ASTO 08
 83 X<>Y
 84 ISG 06

 85♦LBL 15
 86 SF 09
 87 FS? 08
 88 RTN
 89 CF 22
 90 CF 23
 91 FIX 0
 92 CF 29
 93 "#"
 94 ARCL 06
 95 "⊢ OF "
 96 ARCL 07
 97 "⊢?"
 98 XROM "VA"
 99 TONE 7
100 STOP
101 FS? 48
102 GTO 14
103 FC? 22
104 GTO 19
105 GTO 08

106♦LBL 14
107 FC? 23
108 GTO 19
109 XROM "XD"

110♦LBL 08
111 X<0?
112 GTO 03
113 ENTER↑
114 CLA
115 ARCL 08
116 XROM "DC"
117 RCL 06
118 X<=0?
119 GTO 10
120 7
121 MOD
122 X≠0?
123 GTO 07
124 X<>Y
125 RCL 09
126 RCL 10
127 X<> c
128 RCL [
129 STO IND Z
130 X<>Y
131 X<> c
132 R↑
133 RCL 08
134 DSE 09
135 GTO 06
136 ISG 09

137♦LBL 20
138 CF 09
139 CLX
140 X<>Y
141 RCL 07
142 FRC
143 E3
144 *
145 AOFF
146 FIX 0
147 "SST, DEL 00"
148 ARCL X
149 FIX 3
150 XROM "VA"
151 BEEP
152 GTO 00

153♦LBL 01
154 RCL 06
155 X>0?
156 GTO 09

157♦LBL 10
158 "SST, MORE +'S"
159 XROM "VA"
160 TONE 3
161 GTO 00

162♦LBL 03
163 "*CORRECTION*"
164 XROM "VA"
165 TONE 6
166 FC? 09
167 GTO 01
168 DSE 06
169 GTO 14
170 ISG 06
171 GTO 10

172♦LBL 14
173 RCL 06
174 7
175 MOD
176 X=0?
177 ISG 09
178 GTO 14
179 RCL 11
180 X=Y?
181 GTO 13
182 STO 08
183 RDN
184 STO 11
185 GTO 09

186♦LBL 13
187 TONE 4
188 6
189 ST- 06
190 R↑
191 GTO 15
192♦LBL 14
193 CLA
194 ARCL 08
195 ARCL 10
196 CLX
197 X<> \
198 STO [
199 ASTO 08
200♦LBL 09
201 RDN
202 GTO 15

203♦LBL "-B"
204 FC? 08
205 GTO 15
206 FS? 09
207 GTO 08

208♦LBL 19
209 RCL 06
210 X<=0?
211 GTO 10
212 CHS
213 ISG X
214 7
215 MOD
216 X=0?
217 GTO 14
218 CLA
219 ARCL 08

220♦LBL 11
221 "⊢♦"
222 DSE X
223 GTO 11
224 X<> [

225♦LBL 14
226 RCL 09
227 X<>Y
228 RCL 10
229 X<> c
230 X<>Y

231♦LBL 12
232 STO IND Z
233 CLX
234 DSE Z
235 GTO 12
236 RDN
237 X<> c
238 RDN
239 GTO 20

240♦LBL "XD"
241 "⊢♦A"
242 RCL [
243 E2
244 XROM "QR"
245 29
246 ST- Z
247 -
248 .9
249 ST* Z
250 *
251 INT
252 X<>Y
253 INT
254 16
255 *
256 +
257 END
```

TECHNICAL DETAILS

XROM: 10,23 | **L —** | SIZE: 012

Stack Usage:

- 0 T: CLEARED
- 1 Z: CLEARED
- 2 Y: CLEARED
- 3 X: CLEARED
- 4 L: USED

Flag Usage:

- 04: NOT USED
- 05: NOT USED
- 06: NOT USED
- 07: NOT USED
- 08: SET
- 09: SET
- 10: NOT USED
- 25: CLEARED
- 50: CLEARED

Alpha Register Usage:

- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:

- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED BUT RESTORED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 4

ΣREG: UNCHANGED

Data Registers:

- R00: ONLY REGISTERS 6-11 ARE USED
- R06: BYTE NUMBER
- R07: m.00p00q
- R08: BLANK ALPHA STRING
- R09: INDEX FOR BYTES STORAGE
- R10: Rc FOR LOWERED CURTAIN
- R11: CLEARED

Global Labels Called:

Direct	Secondary
VA	2D
VS	PART OF GE
RT	
QR	
OM	

Local Labels In This Routine:

00
06
07
13
14
15

Execution Time: 6.3 seconds (for 20+'s)

Peripherals Required: NONE

Interruptible? YES

Execute Anytime? NO

Program File: LB

Bytes In RAM: 357
(Lines 11 thru 88)

Registers To Copy: 71

Other Comments:

LB - LOAD BYTES

The Load Bytes program LB is a synthetic function assembly routine. It does for synthetic programming what MK did for synthetic function key assignments, enabling the user to key up a program containing synthetic program lines simply by keying in the decimal equivalent of each byte. Normal functions are entered in the ordinary manner.

Unlike MK, LB is not limited to 2-byte functions; synthetic functions and character strings of any length manageable by the 41C firmware can be created. Also, bytes from the lower half of the combined hex table are no more difficult to key in than those from the upper half. Thus, for example, the "critical positions" in the synthetic BLDSPEC strings (see *PPC CALCULATOR JOURNAL,* V7N6P24-28) present no problem at all, nor do the print buffer "control characters" (PPC CJ, V7N6P19-22). Creation of synthetic END's and other lower-half functions is simple, and further experimentation will be greatly facilitated by LB.

More importantly, synthetic programming is brought within reach of the average user, by virtue of the simple, generalized technique. In short, LB entirely eliminates the bother, the limitations and the quirks associated with byte jumping (PPC CJ, V7N6P43-46), "Q" loading (PPC CJ, V7N7P21-25), module pulling (PPC CJ, V7N5P57b), prefix-masking (PPC CJ, V8N1P31d and V8N5P11), and the like. All that is needed is LB, a copy of the combined hex table (PPC CJ, V6N5P22-23) and an understanding of the byte sequences of 41C functions (see Table 1). Recommended for background reading are sections 2A and 2B of Bill Wickes' book and/or the Corvallis Division columns (PPC J, V6N4P11, V6N5P20, V6N6P19). After reading these you should be able to use LB to key up any synthetic program from a printer listing and accompanying documentation.

Example 1: This program demonstrates the use of LB to enter synthetic instructions and how the "goose" may be made to fly backwards. (For more information on the "goose", see PPCJ, V7N5P55.)

1. Key in LBL^{T}GOOSE. 76 bytes of synthetic instructions will be entered (n). We need n' + 6 bytes reserved where n' is the lowest multiple of 7 equal to or greater than n. Here n' = 77 and n' + 6 = 83. Key in LBL^T++ followed by 83 +'s then XROM LB. The XROM should be line 85.

2. Switch out of program mode. R/S. Observe DEC/HEX INPT then #1 OF 77?. We will use hex (alpha). Press Alpha, enter 9C, R/S. Observe #2 of 77. Enter 0A, R/S. (These are the hex codes for FIX A.) Continue until all of the following have been input: (Numbers in parenthesis are to keep track of the number. Remember to press shift before numbers in Alpha.)

```
(3)  F7 01 00 00 00 00 C0 13   (line 6 of program)
(11) F7 01 00 00 00 0C 00 13   (line 8)
(19) F7 01 00 00 00 C0 00 13   (line 10)
(27) F7 01 00 00 0C 00 00 13   (line 12)
(35) F7 01 00 00 C0 00 00 13   (line 14)
(43) F7 01 00 0C 00 00 00 13   (line 16)
(51) F7 01 00 C0 00 00 00 13   (line 18)
(59) F7 01 0C 00 00 00 00 13   (line 20)
(67) F7 01 C0 00 00 00 00 13   (line 22)
(75) CE 75 (line 26, X ≷ M)
```

After each entry observe nn OF 77?. After the last entry observe 77 OF 77? R/S, observe SST, DEL 00n. Press SST, press PRGM, enter DEL 00n.

3. Enter GTO.002 and observe FIX 0 (this is FIX A). Now key in the rest of the program as below. When you get to a synthetic line (06, 08, 10, etc.), just press SST as these are the instructions already entered. After line 28 delete any remaining +'s and the XROM LB. Switch out of program mode, RTN, and R/S. Observe the "goose" headed west instead of east for once.

APPLICATION PROGRAM FOR: LB	
01♦LBL "GOOSE"	16 "x♦μ♦♦♦Ȧ"
02 FIX 0	17 XEQ 99
03 CF 21	18 "x♦♦♦♦Ȧ"
04 CF 28	19 XEQ 99
05 CF 29	20 "xμ♦♦♦♦Ȧ"
06 "x♦♦♦♦Ȧ"	21 XEQ 99
07 XEQ 99	22 "x♦♦♦♦Ȧ"
08 "x♦♦♦μ♦Ȧ"	23 XEQ 99
09 XEQ 99	24 STOP
10 "x♦♦♦♦Ȧ"	
11 XEQ 99	25♦LBL 99
12 "x♦♦μ♦♦Ȧ"	26 X<> [
13 XEQ 99	27 VIEW X
14 "x♦♦♦♦Ȧ"	28 RTN
15 XEQ 99	29 .END.

COMPLETE INSTRUCTIONS FOR LB

1. SIZE 12 or greater is required.
2. Go to the point in program memory where the bytes are to be inserted (need not be at the end of program memory) and key in the following in PRGM mode:
   LBL "++" + + + ... + + + XROM LB
(The +'s serve to make room in program memory for bytes to be loaded, and are also executed for counting purposes, so don't use ENTER or anything else in place of the + instruction.) In order to guarantee room for *n* bytes, the number of +'s should be at least *n'* + 6 where *n'* is the lowest multiple of 7 greater than or equal to *n*. (For example, 13 +'s are sufficient for loading 1-7 bytes, 20 +'s for loading 8-14 bytes, etc.) Or you can simply key in a + for each byte you want to load and then add 12 extra +'s, which will usually result in more +'s than the formula quoted above, but will do no harm unless you are running out of memory. A program register full of unused +'s will automatically get replaced by nulls in the end.
3. There is no advantage to packing if the above lines were keyed in sequentially. If they were not, there may be some nulls among the +'s, and there may be a slight advantage to packing in this case (for convenience in "cleaning up" later--see Steps 7 and 8 below). Packing is not *necessary* in either case, however, and even if you did not key in the filler +'s sequentially you may wish to skip packing if you are in a long program which takes a long time to pack.
4. With the program pointer still at the XROM LB, switch to non-PRGM mode and push R/S. (The contents of the stack are irrelevant at this point, so you could actually push R/S starting at any point between the LBL^T++, +'s and the XROM LB. Or if you have lost your place in program memory, execute LB from the keyboard) After a few seconds the program will stop with the prompt "#1 OF *m*?", where *m* is the maximum number of bytes that can be loaded considering the amount of +'s you inserted in Step 2.

LB

5. If the displayed m is not large enough for your needs, execute GTO"++" and insert more +'s--preferably a multiple of 7, which will avoid intervening nulls. If you didn't put in enough +'s to load any bytes at all, you will get the message SST, MORE +'s, at which point pushing SST once will get you to the LBL^T++ where you can insert more +'s. After inserting the +'s switch out of PRGM mode and push R/S to restart the byte-loading process.

6. Key in either the decimal equivalents of the bytes in non-ALPHA mode, or the hexadecimal digits of the bytes in ALPHA mode, pushing R/S after each byte. Decimal and hex entries may be freely mixed; each entry is interpreted as decimal or hex depending on whether the calculator is in non-ALPHA or ALPHA mode at the time R/S is pushed. For hex entries, two and only two digits must be entered (including leading zeros). Incidentally, the decimal equivalent of the byte entered remains in the X-register when the next prompt appears, but to repeat a byte you must actually key in the number again, or else set Flag 22 (see below).

7. To terminate the byte-loading process, just push R/S without making any entry. This may be done in either ALPHA or non-ALPHA mode. If you have loaded in as many bytes as there was room for (i.e., made an entry after seeing "*#m* OF *m?*"), the byte loading will be terminated automatically. In either case you will see the prompt "SST, DEL 00p", where p is a number from 1 to 7. Push SST once, then go into PRGM mode (where you should see LBL^T++) and execute DEL 00p. This will get rid of the LBL^T++ and the 0 to 6 initial +'s that were in the same program register as the label. (If there were extra nulls among the +'s any you did not choose to pack in Step 3, then there may be more than p lines to delete, but the extra +'s can be back-arrowed separately.) You may now SST through the synthetic lines that you have created.

8. After your newly-created synthetic lines there will be from 0 to 6 final +'s (the rest were automatically replaced by nulls when the byte loading was terminated), followed by the XROM **LB** instruction. If you are going to do more byte loading in the same program, you could leave these instructions in for future use (but remember to get rid of them before running your program). Otherwise you might as well delete them now. For convenience, the number of final lines to be deleted can be found by looking at the fractional part of the number left in the X-register (the integer part is the number p referred to in Step 5). In other words, X will contain a number of the form p.00q; go to the first + after your synthetic lines and execute DEL 00q. (If, however, the number of initial lines to delete was more than p due to unpacked nulls, then the number of final lines to delete may be less than q, in which case it would be easiest to back-arrow the lines one at a time.)

9. If you find you have made an error in a byte entry and have already pushed R/S, simply push XEQ 03 (note that the 03 can be obtained by the C key). Or if you are in ALPHA mode, just enter the character C (actually any single character will do) and push R/S. The last byte entered will be effectively deleted, and the byte number in the prompt decremented by 1. You can back up at least 7 bytes in this way, but after some point (ranging from 7 to 13 bytes of backing up) further backing up will be in multiples of 7 bytes until the beginning is reached. (For example, if you are being prompted for byte #24 and decide to XEQ 03 repeatedly, you will be prompted for bytes #23, 22, 21, 20, 19, 18, 17, 16, 15, 8, 1, 1, 1, ...) Even if you have terminated the byte-loading process, or if it has been terminated automatically, you can still back up and make changes using XEQ 03 as long as you have not pushed SST and left the **LB** program. (Note: Entering a negative number for a byte entry has the same effect as XEQ 03; this feature was included mainly for use with the promptless version.) The number in X is preserved during the correction process, in case you want to refer to the last byte entered (except when error correction is made after a termination, in which case the number p.00q is in X and the last-entered byte in Y).

10. If you have lost track of what byte you are on and want a reprompt, push XEQ 01. (If you have terminated the byte loading, but not yet pushed SST, then XEQ 01 has the same effect as XEQ 03.)

WARNINGS: (a) It is up to the user to plan ahead so that there is enough room to finish multibyte lines. If you start on a 15-character text line with only 2 bytes left, and terminate the loading process without correcting this error, you may find some of the rest of your program (including vital END's, etc.) gobbled up by the lext line, and consequent problems with the global label linkage. (b) Don't pack or change size during the byte loading process. (c) Don't leave the byte-loading program without using the termination procedure (Step 7), especially if you have made error corrections, or you may end up with bytes missing or extra unwanted bytes. (d) During byte loading you may use the stack freely for calculations, and also registers 00-05, but don't store anything in registers 06-11, as these registers contain vital information used by the program. (R_{06} = byte #; R_{07} = maximum byte #, actually the number m.00p00q where m, p, and q are as defined above; R_{08} = partial register of bytes; R_{09} = index for indirect storage of bytes in program memory; R_{10} = c-register contents for lowered curtain; R_{11} = previously-stored 7 bytes for backup purposes.) Also avoid changing Flags 08 and 09. One should also be aware of the fact that Flags 22 and 23 are used to detect whether an entry has been made: Flag 22 only is tested if the calculator is non-ALPHA mode, while Flag 23 only is tested if in ALPHA mode.

LINE BY LINE ANALYSIS OF LB

Upon reading through **LB**, the reader will notice one unusual feature: the program does not start with a global label, but has LBL 00 STOP GTO "++" as its first three lines. This is because the GTO "++" line is where the program pointer is when the program steps with the SST, DEL 00p or SST, MORE +'s prompt, and the SST operation takes less time to execute the closer the pointer is to the beginning of the program (due to the fact that the line number is calculated).

When **LB** is run in its normal prompting version the program is entered via the label in line 04, either by pushing R/S with XROM **LB** as the instruction in memory or by executing **LB** from the keyboard. Line 04 is actually reached twice during the course of program operation, the first time with no message in the display (flag 50 clear), and later on with the DEC/HEX INPUT message in the display (flag 50 set). The first time line 06 is skipped, and lines 07 through 10 and 22 through 32 put the above mentioned message in the display, clear flag 08 to signal the prompting version, check for sufficient size, initialize register 06, and arrange the stack with T = Z= Y=1, X = 0.

At line 33 we branch to LBL "++" which the user is supposed to have put in program memory, and the +'s are

Routine Listing For: **LB**

```
01♦LBL 00
02 STOP
03 GTO "++"

04♦LBL "LB"
05 FS? 50
06 GTO 00
07 "DEC/HEX INPT"
08 XROM "VA"
09 CF 08
10 GTO 13

11♦LBL "L-"
12 CLA
13 XROM "VA"
14 CF 08
15 RCL a
16 STO [
17 RCL b
18 FS? 08
19 GTO 14
20 STO \
21 SF 08

22♦LBL 13
23 12
24 XROM "VS"
25 FC?C 25
26 PROMPT
27 CLST
28 STO 06
29 SIGN
30 ENTER↑
31 ENTER↑
32 R↑
33 GTO "++"

34♦LBL 00
35 RCL b
36 FC? 08
37 GTO 14
38 CLD
39 X<> [
40 STO a
41 X<> \
42 X<> b

43♦LBL 14
44 XROM "RT"
45 117
46 X<>Y
47 -
48 7
49 XROM "QR"
50 ST- Z
51 X<>Y
52 CHS
53 STO 09
54 X<> Z
55 LASTX
56 XROM "QR"
57 1.001
58 ST* 09
59 ST+ Y
60 FRC
61 ST* T
62 X<>Y
63 R↑
64 +
65 *
66 X<>Y
67 X<=0?
68 GTO 10
69 ST+ 09
70 7
71 *
72 +
73 STO 07
74 XROM "OM"
75 X<> c
76 STO 10
77 CLST

78♦LBL 06
79 STO 11
80 CLA

81♦LBL 07
82 ASTO 08
83 X<>Y
84 ISG 06

85♦LBL 15
86 SF 09
87 FS? 08
88 RTN
89 CF 22
90 CF 23
91 FIX 0
92 CF 29
93 "‡"
94 ARCL 06
95 "├ OF "
96 ARCL 07
97 "├?"
98 XROM "VA"
99 TONE 7
100 STOP
101 FS? 48
102 GTO 14
103 FC? 22
104 GTO 19
105 GTO 08

106♦LBL 14
107 FC? 23
108 GTO 19
109 XROM "XD"

110♦LBL 08
111 X<0?
112 GTO 03
113 ENTER↑
114 CLA
115 ARCL 08
116 XROM "DC"
117 RCL 06
118 X<=0?
119 GTO 10
120 7
121 MOD
122 X≠0?
123 GTO 07
124 X<>Y
125 RCL 09
126 RCL 10
127 X<> c
128 RCL [
129 STO IND Z
130 X<>Y
131 X<> c
132 R↑
133 RCL 08
134 DSE 09
135 GTO 06
136 ISG 09
```

Routine Listing For: **LB**

```
137♦LBL 20
138 CF 09
139 CLX
140 X<>Y
141 RCL 07
142 FRC
143  E3
144 *
145 AOFF
146 FIX 0
147 "SST, DEL 00"
148 ARCL X
149 FIX 3
150 XROM "VA"
151 BEEP
152 GTO 00

153♦LBL 01
154 RCL 06
155 X>0?
156 GTO 09

157♦LBL 10
158 "SST, MORE +'S"
159 XROM "VA"
160 TONE 3
161 GTO 00

162♦LBL 03
163 "*CORRECTION*"
164 XROM "VA"
165 TONE 6
166 FC? 09
167 GTO 01
168 DSE 06
169 GTO 14
170 ISG 06
171 GTO 10

172♦LBL 14
173 RCL 06
174 7
175 MOD
176 X=0?
177 ISG 09
178 GTO 14
179 RCL 11
180 X=Y?
181 GTO 13
182 STO 08
183 RDN
184 STO 11
185 GTO 09

186♦LBL 13
187 TONE 4
188 6
189 ST- 06
190 R↑
191 GTO 15

192♦LBL 14
193 CLA
194 ARCL 08
195 ARCL 10
196 CLX
197 X<> \
198 STO [
199 ASTO 08

200♦LBL 09
201 RDN
202 GTO 15

203♦LBL "-B"
204 FC? 08
205 GTO 15
206 FS? 09
207 GTO 08

208♦LBL 19
209 RCL 06
210 X<=0?
211 GTO 10
212 CHS
213 ISG X
214 7
215 MOD
216 X=0?
217 GTO 14
218 CLA
219 ARCL 08

220♦LBL 11
221 "├♦"
222 DSE X
223 GTO 11
224 X<> [

225♦LBL 14
226 RCL 09
227 X<>Y
228 RCL 10
229 X<> c
230 X<>Y

231♦LBL 12
232 STO IND Z
233 CLX
234 DSE Z
235 GTO 12
236 RDN
237 X<> c
238 RDN
239 GTO 20

240♦LBL "XD"
241 "├♦Δ"
242 RCL [
243  E2
244 XROM "QR"
245 29
246 ST- Z
247 -
248 .9
249 ST* Z
250 *
251 INT
252 X<>Y
253 INT
254 16
255 *
256 +
257 END
```

executed, causing the total number of +'s to end up in X. Then the XROM **LB** after the +'s is executed, sending us to line 04 of **LB** for the second time and also providing us with a return address (in the b-register) indicating where in program memory the XROM **LB** was. This time line 06 is executed, sending us to line 34 through 36 and 43 through 44 where we recall the b-register and use routine RT to obtain the decimal-equivalent of the return address, that is, the decimal address of the byte to which the program pointer would be sent to if a RTN were executed. The byte so addressed is the last byte of the 2-byte XROM LB instruction in program memory. Let $\underline{r}$ be this decimal address (in X) and $\underline{S}$ the number of +'s (in Y). The address of the last + is r + 2, and the number of +'s in the register which contains the beginning of the XROM LB instruction is (-r-2) mod 7. [These +'s and the XROM LB will be left-over after the byte loading is completed, so that the number of lines to be deleted at the end is q = 1 + (-r-2) mod 7, which can range from 1 to 7.] The absolute address of the register immediately above this register, which is the last register of +'s into which bytes can be stored, is the lowest integer greater than or equal to (r + 2)/7, or equivalently, -[-(r + 2)/7] where [] denotes the greatest-integer function in the algebraic sense; e.g. [-2.1] = - 3. For indexing purposes (using DSE) we need one less than this address relative to a curtain address of 16, that is,

$$a_f = -[-(r + 2)/7] -17$$
$$= -[(117 - r)/7]$$

Also, since (-r-2) mod 7 = (117 - r) mod 7, it is evident that the number of final +'s and the address a_f can be obtained by applying routine QR with 117 - r in Y and 7 in X. This is done in lines 45 through 49. (Note: If LB is copied and used in RAM, then instead of the 2-byte XROM LB we have a 4-byte XEQ LB. For the proper counting of bytes the quantity r + 2 should be replaced by r + 4, and hence the number 117 replaced by 115, as it is in the RAM-version program described and bar-coded in PPC CJ, V8N2P34-40.)

At this point we have Z = S, the total number of +'s; Y = $-a_f$, the negative of the address of the register containing the final +'s (relative to register 16); X = q - 1, the number of final +'s. If the quantity Z - X is divided by 7 the integer quotient will be the number of registers (call it k) into which bytes can be stored, and the remainder will be the number of initial +'s (p - 1 using the notation of the instructions). This is accomplished in lines 50 and 54 through 56, while a_f is also stored in R09 (lines 51 through 53). After line 56 we have T = a_f, Z = q - 1, Y = k, X = p - 1. Lines 57 - 65 yield R09 = 1.001 * a_f, T = Z = Y = k, and X = .001 * (p + .001 * q) (which has the digital form .00p00q). In lines 66 through 68, k is checked for positiveness; if it is not we branch to LBL 10 (line 157) to display the SST, MORE +'s message and stop (lines 158 through 161 and 01 through 03). If k is positive we go on to line 69 whence R09 becomes (a_f + k) + a_f/1000, which is just the right form iii.fff of the index for storing into registers a_f + k, a_f + k - 1, ..., a_f + 1. Also, we find 7 * k = m, the maximum number of bytes that can be loaded, and obtained in R07 the number m.00p00q (lines 70 through 73). Finally, we create and store the number to be placed in the c-register to lower the curtain (lines 74-76) and initialize the stack and other registers (lines 77 through 84) as we enter the main byte-loading loop.

The usage of the data registers is as follows:

R06 = Byte number (integer)
R07 = m.00p00q
R08 = 0 to 6 bytes already assembled, but not yet stored in program registers.
R09 = Index for storing into program registers.
R10 = Number to be stored in c-register to lower curtain to 16.
R11 = First 6 bytes of previously-stored program register, saved here to enable backing up at least 7 bytes.

The byte-loading loop actually begins at LBL 07 (line 81), where the bytes assembled so far are stored in R08 and the byte number incremented (lines 82 through 84), unless the byte number is a multiple of 7, in which case the loop begins farther up at LBL 06 (line 78) to additionally store the backup bytes and initialize R08. For a correction or reprompt (in which the byte number is not incremented) the loop begins at LBL 15 (line 85). Incidentally, in the interest of avoiding accidental execution of labels, none of the labels 01 through 05 are used except for LBL 01 (executed for reprompt) and LBL 03 (executed for correction). This required using a few 2-byte labels (15, 19 and 20), but all branches to these labels are long enough to require 3-byte GTO's anyway. Many other branches to 1-byte labels have been made synthetic 3-byte GTO's in order that all GTO's will be compiled.

Before proceeding further, let us look at what happens when L- is executed, starting at line 11. This line is reached only once during the course the program, as the control goes to LBL LB after the +'s in RAM have been executed. Lines 12-13 serve to set flag 50 for proper action later. There is an extra matter to be dealt with, however: since L- is to be called as a subroutine by the user's program, it is necessary to preserve the subroutine return address(es) in registers a and b--we do not want the return address produced by the XROM LB in RAM (which we use to determine where the bytes are to go by the process described above) to be added to the subroutine return stack. We accordingly save registers a and b, storing them temporarily in M and N (lines 14 through 18, 20 and 21). Flag 08 is temporarily used to control the behavior after the RCL b of line 17, ends up set and serves to signal the non-prompting version of the byte loader. We then go to lines 22 through 33, 04 through 06, and 34 through 36 as with LB. However, the set status of flag 08 causes line 37 to be skipped and lines 38 through 42 executed, which restores the a- and b- registers. Once the b-register is restored, the program pointer is suddenly back to where it was when b was recalled, namely at line 18, whereupon the setting of flag 08 causes a branch at line 19 to line 43 where we rejoin the path taken by LB.

After LBL 15 (line 85) we set flag 09 for control purposes (see below where the correction sequence is described), and in the case of the non-prompting version return to the program which called it (lines 87 and 88). In the case of the prompting version we proceed to clear the data-entry flags and prompt for a byte (lines 89 through 100). Flag 48, the alpha-mode flag, is checked in line 101. If it is clear (indicating non-alpha mode) then flag 22 is checked to see if there was an entry--if not, a branch is made to LBL 19 (line 208) to terminate the byte-loading. If flag 48 was set, lines 106 through 109 check flag 23 to see if an alpha (hexadecimal) entry was made, using XD to convert it to decimal if so, and branching to LBL 19 if not. At LBL 08 (line 110) we have the decimal byte in X; lines 111-112 branch to the back-up (correction) routine if X is negative. Lines 113 through 116 copy the decimal byte for future reference and use DC to append the byte to the bytes from R06 waiting to be loaded. Lines 117 through 123 check to see if the byte number is a multiple of 7; if not we branch back to LBL 07 (line 81) to store the partially-assembled bytes and prompt for a new byte (lines 118 and 119 are simply a guard against illegal data in R06 in case things were not initialized properly.) If the byte number is a multiple of 7, the seven bytes in alpha are stored in the appropriate program register (lines 124 through 131) by lowering the curtain, the first six bytes are recalled (line 132 and 133) to be later stored in R11, the storage index in R09 is decremented (line 134), and we go back to LBL 06 (line 78 where the back-up bytes are placed in R11 and the loop begun again. If, however, the last register of +'s was used, then the DSE in line 134 causes a skip, sending us to lines 136 through 152 and 01 through 03 where the byte loading is terminated automatically and the SST, DEL 00p prompt displayed. This sequence is also joined at LBL 20 (line 137) when the byte-loading is terminated by pushing R/S without an entry; recall that this sent us to LBL 19 (line 208). Lines 209 through 217 decide whether the byte number is a multiple of 7, in which case the bytes have already been stored in the program registers. If not, enough nulls are appended to fill a 7-byte register and the result put in X (lines 218 through 224). Lines 225 through 239 fill the remaining program registers with first the bytes in X and then nulls, finally branching to LBL 20 (line 137) to display the SST... prompt as mentioned above. (The loop in lines 231 through 235 is executed with the curtain down, incidentally, so that if this program is copied into RAM it must have and END before it to execute properly.)

Lines 153 through 156 and 200 through 202, starting with LBL 01, implement the reprompt feature. The sequence starting with LBL 03 (line 162) takes care of corrections. Flag 09 is always set (line 86) before prompting for a new byte (or returning, in the non-prompting case), but is cleared (line 138) upon termination of the byte loading. The correction sequence makes use of this information in lines 166 and 167 to decide whether or not to decrement the byte number. The net result of lines 168 through 171 is to decrement the byte number, unless it is already 1 (in which case it stays 1) branching to LBL 10 only if the byte number somehow was zero or negative (indicated that R06 was uninitialized or tampered with). There are three possible cases of backing up: (a) The byte number (after being decremented) is not a multiple of 7, in which case line 177 is skipped, line 178 executed, and lines 192 through 199 executed to remove the last byte from the bytes stored in R08. (Line 195, ARCL 10, is simply to perform a 6-byte shift.) (b) The byte number is a multiple of 7 and R11 contains backup bytes, in which case line 177 is executed to back up the storage index, line 178 is skipped and lines 179 and 180 and 182 through 185 cause the backup bytes from R11 to be stored in R08, and R11 to be cleared. (At line 180, Y = 0 and X = R11, so the answer is "false".) (c) The byte number is a multiple of 7 and R11 = 0, i.e., there are no more backup bytes available, in which case line 180 tests "true" and line 181 sends us to lines 186 through 191 where an extra tone (Tone 54) is sounded and the byte number is further decremented by 6, resulting in a 7-byte backup. This is necessary because once a register of bytes has been stored we cannot recall it to change some of the bytes, but rather the whole register must be composed and stored again, requiring a 7-byte backup unless the bytes were stored somewhere else where they <u>can</u> be accessed (hence the purpose of R11, which allows at least 7 one-byte backups before 7-byte backups become necessary).

Finally, routine **B** starts at line 203. Lines 204 and 205 are mainly a check of flag 08, which should be set because **L** is supposed to have been executed first. If flag 08 is clear, then **B** serves as a reprompt function, sending us back to LBL 15. Assuming the normal case of flag 08 is set, however, we go on to lines 206 and 207 which send us to LBL 08 (line 110) to process the decimal byte in X if flag 09 is set, and pass us on to the termination sequence beginning with line 208 if flag 09 is clear. (The number in X is not processed in the latter case.)

The reader who is thoroughly confused with all or part of the above analysis (particularly with the relations between k, m, p, q, r and s discussed earlier) is encouraged to follow through the portions of the program for specific examples and verify that they work, after which an overall understanding is more likely to emerge.

CONTRIBUTORS HISTORY FOR LB

The idea of a special-purpose routine for storing arbitrary synthetic codes directly into program memory apparently originated with Bill Wickes (3735) and his B2 program (see *PPC CALCULATOR JOURNAL*, V7N3P7). [Earlier John McGechie (3324) had written a program to create the synthetic lines needed by his KA #12 program. This was probably the first time that instructions were created under program control.] B2 required the user to execute a "CODE" (**HN**) routine to create the desired bytes, then specify the hexadecimal address of the register in which the code was to be stored. John McGechie made this procedure more convenient with his b2 program (see *PPC TECHNICAL NOTES*, V1N3P29), which accepted the destination address in program pointer form. Thus, the user had only to create the 7 byte code, RCL b at the desired location, and XEQ "b2".

User convenience in byte loading took a giant step forward when William Cheeseman (4381) wrote ASP (Assembler for Synthetic Programs). He sent copies to several PPC members, kicking off the quest for the ultimate byte loader. ASP accepted input in the form of decimal codes (0-255) for each byte. The codes were assembled into seven byte groups and stored in data registers, after which the curtain was raised to convert the stored data to program steps.

Bill Wickes (3735) and Keith Jarett (4360) each responded to ASP with programs that stored the 7 byte groups directly into program memory. This step in the evolution effectively combined the best of the first two approaches, in that it accepted a RCL b program pointer to designate the destination address and in that input was in the form of individual decimal byte codes. Bill Wickes's SYN was an overnight success, and Keith Jarett and Bill Cheeseman collaborated in developing the first published version of **LB** (see *PPC CALCULATOR JOURNAL*, V7N10P20). This version incorporated several suggestions from Roger Hill (4940), the main one of which was to decode a subroutine return address to determine the destination of the bytes, eliminating the need for a RCL b assignment.

Roger Hill spent a considerable amount of time and effort in developing **LB** for the PPC ROM. Added features included automatic counting of bytes (suggested by Richard Nelson (1)), protection from over-writing other lines, a hexadecimal input option, extended correction capability, and prompting for the cleanup of unused "buffer" bytes. LB is another one of Roger's masterpieces and is probably the ultimate byte loader.

FINAL REMARKS FOR LB

Table 1 is a short description of the various types of instructions that may be loaded into program memory using LB. This table is a convenient "quick reference" for everyday use.

TABLE 1

41C Program Instruction Byte Structure:

(references to combined hex table (PPC J, V6N5P22-23)

1. One-byte function: byte 1 from rows 1-8 and special cases. E.g., the "ultimate NOP" (PPC CJ, V7N3P30) = 240 (text 0).
2. Two-byte function: byte 1 from rows 9-B and bytes 206 and 207; byte 2 from postfix part of any row (top half of table for direct execution, bottom for indirect). E.g., STO M = 145, 117; STO IND M = 145, 245; local LBL M = 207, 117; STO 100 (display shows STO 00) = 144, 100; RCL 111 (display shows RCL j) = 145, 111.
3. Character string: byte 1 from row F; subsequent bytes from any row (only top half of table prints). E.g., "A#" = 242, 65, 35.
4. Append character string: byte 1 from row F (use text byte 1 unit higher than desired number of characters); byte 2 is 127; subsequent bytes from any row (only top half of table prints). E.g., "⊦A#)" = 244, 127, 65, 35, 41.
5. Alpha (global) label: byte 1 is 192; byte 2 is 0; byte 3 from row F (use text byte 1 unit higher than desired number of characters); byte 4 is 0; subsequent bytes from any row (only top half of table prints). E.g., LBL "A#)" = 192, 0, 244, 0, 65, 35, 41; global LBL "A" = 192, 0, 242, 0 65.
6. Alpha (global) GTO or XEQ: byte 1 is 29 or 30; byte 2 from row F; subsequent bytes from any row (only top half of table prints). E.g., GTO "A#)" = 29, 243, 65, 35, 41.
7. Numeric (local) GTO or XEQ: short-form GTO, byte 1 from row B, byte 2 is 0; XEQ or long-form GTO, byte 1 from row E or D, byte 2 is 0, byte 3 from postfix part of any row (direct execution only); GTO IND or XEQ IND, byte 1 is 174, byte 2 from postfix part of any row (top half of table for GTO IND, bottom for XEQ IND). E.g., GTO 01 = 178, 0; GTO 99 = 208, 0, 99; XEQ IND 99 = 174, 227; local GTO M = 208, 0, 117.
8. Short-form exponent: byte 1 is 27; byte 2 from row 1, columns 0-9; byte 3 (for 2-digit exponents) likewise. For negative exponent, insert 28 (not 84) after 27. E.g., E3 = 27, 19; E99 = 27, 25, 25; E-50 = 27, 28, 21, 16.
9. Number entry: row 1, columns 0-C. Successive bytes will extend a single program line (i.e., create a multi-digit number). Use 0 (null) or 131 (ENTER↑) to terminate digit entry before starting a new program line consisting of another number. Use 28 (not 84) prior to digit bytes for negative numbers. E.g., - 1.75E-10 = 28, 17, 26, 23, 21, 27, 28, 17,16.
10. XROM: see PPC CJ, V7N7P10. E.g., VER (XROM 30, 05) = 167, 133. Also see XL .

FURTHER ASSISTANCE ON LB

Call William Cheeseman (4381) at (617) 235-8863.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,22 | **LB** | SIZE: 012

Stack Usage: *ON TERMINATION
- 0 T: USED
- 1 Z: USED
- 2 Y: PREVIOUS ENTRY *
- 3 X: PREV. ENTRY OR p.00q*
- 4 L: USED

Flag Usage:
- 04: NOT USED
- 05: NOT USED
- 06: NOT USED
- 07: NOT USED
- 08: NON-PROMPTING VERSION
- 09: USED
- 10: NOT USED
- 22: NUMERIC (DECIMAL) ENTRY
- 23: ALPHA (HEX) ENTRY
- 25: CLEARED 29: CLEARED

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O: ALL USED
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: USED
- 12 b: USED
- 13 c: USED BUT RESTORED
- 14 d: USED
- 15 e: NOT USED

Display Mode: FIX 3

Angular Mode: UNCHANGED

Unused Subroutine Levels: 0

ΣREG: UNCHANGED

Data Registers:
- R00: ONLY REGISTERS 6-11 ARE USED
- R06: BYTE NUMBER
- R07: m.00p00q
- R08: PARTIAL REGISTER OF BYTES
- R09: INDEX FOR BYTES STORAGE
- R10: Rc FOR LOWERED CURTAIN
- R11: PREVIOUSLY-STORED 7 BYTES
- R12: NOT USED

Global Labels Called:

Direct	Secondary
VA	2D
VS	PART OF GE
RT	QR
QR	
OM	
XD	
DC	

Local Labels In This Routine: 00, 06, 07, 09, 11, 12, 13, 14, 15
- 01 REPROMPT
- 03 CORRECTION
- 08 PROCESS DECIMAL INPUT
- 10 MORE +'s
- 19 BEGIN TERMINATION
- 20 TERMINATION PROMPT

Execution Time: 7.1 seconds for first PROMPT (20+'s); 2.5 sec. for each decimal input; 3.6-3.9 per HEX input.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: LB
Bytes In RAM: 449
Registers To Copy: 71

Other Comments: Must be preceded by an END if copied into RAM.

LF - LOCATE FREE REGISTER BLOCK

LF is used by the key assignment programs A?, F?, MK, and 1K (and, occasionally, +K) to find out where the key assignments leave off and how much room there is up to the .END. of program memory. As shown in Figure 1, the key assignments begin at location 192 (decimal) and move upward into the unused area while program memory runs downward from the "curtain" to the .END.. When more program memory is needed, the .END. is moved down one regiser into the unused area.

Status register c contains a pointer to the .END., so locating it is easy (see E?). However, no pointer is maintained to indicate the location of the uppermost key assignment register in use. LF must therefore recall key assignment registers one by one until it finds an empty one. LF finishes with an ISG counter corresponding to the free register block, except that 16 is subtracted from both the beginning and ending absolute register addresses.

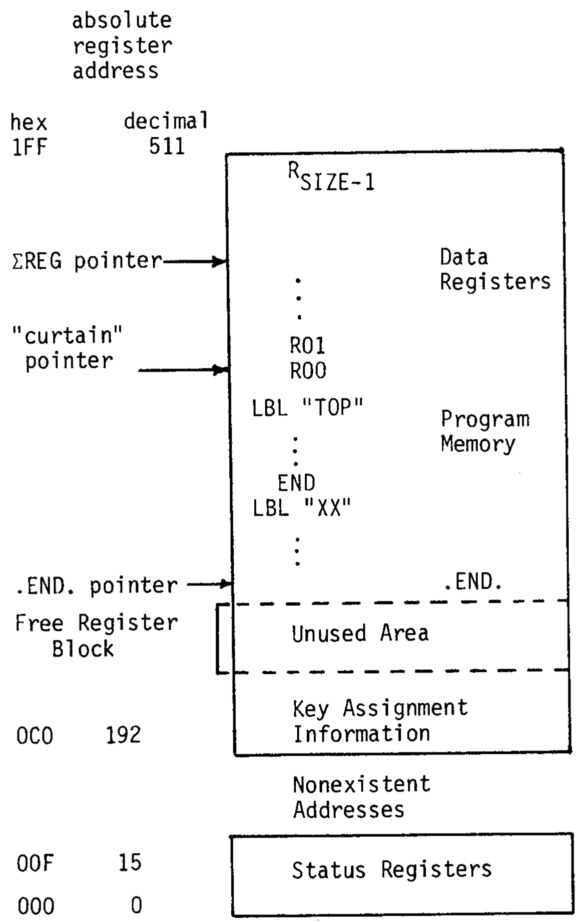


Figure 1. HP-41C Memory Map

Example 1: After MASTER CLEAR, XEQ LF returns a value of 176.221. This indicates that the first free register is located at 176 + 16 = 192, while the last is at 221 + 16 = 237. The total number of free registers is 221 - 175 = 46, in agreement with the PRGM mode display.

COMPLETE INSTRUCTIONS FOR LF

Just XEQ LF to get the bbb.eee counter corresponding the block of free registers. bbb and eee are the addresses, relative to a curtain address of 16 decimal, of the beginning and end of the block of free registers between the key assignments and the .END. of program memory. Y contains a temporary c-register (with curtain = 16) created by OM. If the topmost key assignment register has only its left half used, that register is considered part of the free block and flag 10 is set. In this case the key assignment in that half-register appears in the first 6 characters of alpha for use by MK.

APPLICATION PROGRAM 1 FOR LF

The following program, PRK, is actually a modification of LF. When executed, PRK will decode, print, and restore the contents of the key assignment registers. The printing is in double-width mode with the initial F0 byte omitted. The registers are printed from the bottom (starting with absolute address 192) on up. PRK stops when an empty register or the .END. is reached. As with LF, the program should not be abandoned in the middle (unless you can stop at the beginning of a loop and restore the curtain manually) or the key-assignment registers may be disrupted. Also, this program cannot be the first program in user program memory, but must be preceded by at least one END, because it involves a backward GTO with the curtain lowered.

To construct "PRK" from LF, proceed as follows:

(1) Copy the LF group into user program memory. (You will need at least 59 free program registers for this, but only temporarily.)

(2) GTO.039 and DEL 999 to get rid of everything from line 39 on.

(3) After line 38 key in CF 12 ADV.

(4) Delete lines 29-35 by keying in GTO.029, DEL 007.

(5) After line 24 (RCL N) insert: XROM NH AOFF PI STO P STO 0 ASHF ASHF CLX X<>Y SF 12 PRA.
(NOTE: The decimal codes for STO P and STO 0 are 145, 120, and 145, 119; they can be used in MK to assign those two functions to keys, or in LB to load the functions directly into program memory.)

(6) Delete line 23 (RDN) and replace by R↑ R↑.

(7) After line 18 (GTO 14), insert a 7-character text line whose bytes (in hexadecimal) are:
F7 10 2A 00 00 2A 2A F0.
(This can be easily done using LB.)
(Line 19 should look like ᵀ⊠ * ‾ ‾ * * ⊠, or after SST and PRA one should see θ * ♦ ♦ * * p).

(8) After line 12 (ENTER↑), insert CF 10.

(9) Delete line 10, an 8-character text line.

(10) After line 02 (XROM E?) insert PRA.

(11) Change Line 01 (LBL LF) to LBL "PRK".

(12) Execute GTO.. to pack and put an END on the program. You should have the 48-line 92-byte program shown below:

APPLICATION PROGRAM FOR: **LF**

```
01♦LBL "PRK"          25 R↑
02 XROM "E?"          26 R↑
03 PRA                27 RCL \
04 17                 28 XROM "NH"
05 -                  29 AOFF
06 E3                 30 PI
07 /                  31 STO ↑
08 177                32 STO ]
09 +                  33 ASHF
10 XROM "OM"          34 ASHF
11 .                  35 CLX
12 ENTER↑             36 X<>Y
13 CF 10              37 SF 12
14 DSE T              38 PRA
15 GTO 14             39 X<> IND T
                      40 ISG T
16♦LBL 00             41 GTO 00
17 X<> IND T
18 X=Y?               42♦LBL 14
19 GTO 14             43 X<> Z
20 "θ*****"           44 X<> c
21 X<> [              45 R↑
22 "├θ"               46 CF 12
23 STO \              47 ADV
24 ARCL X             48 END
```

Replace PRA by AVIEW or PROMPT for a non-printer version.

Routine Listing For: **LF**

```
01♦LBL "LF"                          28♦LBL 14
02 XROM "E?"      15♦LBL 00          29 X<> [
03 17             16 X<> IND T       30 ARCL X
04 -              17 X=Y?            31 X<> \
05 E3             18 GTO 14          32 SF 10
06 /              19 X<> [           33 X=Y?
07 177            20 "├θ"            34 DSE T
08 +              21 STO \           35 CF 10
09 XROM "OM"      22 ARCL X          36 X<> Z
10 "*θ*****"      23 RDN             37 X<> c
11 .              24 RCL \           38 R↑
12 ENTER↑         25 X<> IND T       39 RTN
13 DSE T          26 ISG T
14 GTO 14         27 GTO 00
```

Line 10: F8 2A 10 2A 00 00 2A 2A F0

LINE BY LINE ANALYSIS OF LF

The basic plan of the routine is to recall each key assignment register, starting with absolute decimal address 192 and continuing until either an empty register is encountered or the .END. is reached, whichever comes first. To do this the curtain is set at 16 by OM ; relative to this curtain the lowest assignment register is 192-16 or 176. There is a slight complication, however, in that recalling a data register causes both the recalled number and the number in the register to be normalized. Fortunately, the first byte is always F0, which becomes 10 after normalization without any other bytes being changed, so by changing the 10 back to F0 and re-storing it, the key assignment register contents will be rendered "as good as new".

The routine first calculates the .END. address (line 02) and after line 08 has in X the number 177.eee where eee is the number of the register just below the .END., relative to a curtain address of 16. The curtain is lowered in line 09, some useful bytes put into alpha in line 10 (see below), and after line 13 we have T = 176.eee, Z = old R_c contents, Y=X=0. Now, if eee = 175 (occurring if there is no room for key assignments at all) then line 14 is executed and we branch to the concluding sequence. Otherwise line 14 is skipped and we enter the key-assignment register search loop, lines 15-27.

The search loop begins by getting an assignment register (line 16), temporarily storing zero in that register. If that register is zero (line 17) then the search is over and we branch (line 18) to the concluding sequence. Otherwise, we change the first byte to F0 (lines 19-24) and put the result back into the key assignment register, bringing zero back to X (line 25). We then increment the register search index in T and go through the loop again unless the .END. has been reached, in which case we enter the concluding sequence (line 28-39).

In the concluding sequence we shift the contents of the last key assignment register (which was in N) so as to isolate the last byte, which ends up in X followed by six null bytes (lines 29-31). If this byte is zero (line 33), which is the case if the right half of the last assignment register is "void", then in line 34 the index in T is decremented (to indicate one more partly-free register) and line 35 is skipped so that flag 10 ends up set. Otherwise, T is not decremented and flag 10 ends up clear. Finally we restore the original curtain and rearrange the stack so that X = bbb.eee and Y = R_c contents for lowered curtain; both of these are used by MK as is the setting of flag 10 and the contents of the alpha register. If flag 10 is set then the first 6 non-null-characters of alpha are 2A 2A F0 a_1 a_2 a_3--where a_1, a_2, and a_3 are the bytes of the assignment in the "left half" of the top key assignment register--ready to be ASTO'd by MK (2A is simply a filler asterisk).

Note the purposes served by the 8 bytes put into alpha in line 10. Register N contains 6 nulls and an asterisk whose sole purpose is to give a "no" response to line 33 in the case where the search loop was never entered. Register M contains 10 2A 00 00 2A 2A F0 which is made use of as follows: (1) the F0 is grafted onto the recalled key assignments in line 21 to give the proper initial byte, (2) by ARCL-ing this number from X we obtain a 6-byte shift (lines 22 and 30), and (3) after line 30 these bytes have gotten pushed into the O and P registers; the two null bytes then appear as the beginning of the alpha string and the following 6 bytes are 2A 2A F0 a_1 a_2 a_3 as described above. Note also that the 7-byte sequence replenishes itself in M due to line 20 (which appends the byte 10) and 22 (which appends the other 6 bytes)!

CONTRIBUTORS HISTORY FOR LF

LF was written by Roger Hill (4940), who first wrote a similar number-of-assignment-registers in the summer of 1980 which lacked a few refinements of the present routine (such as being interruptable without memory loss or label linkage loss).

FINAL REMARKS FOR LF

Other comments:

1. See remarks in "Background for MK " about extraneous data between the key-assignment registers and the .END.

2. Although LF may be interrupted and/or single-stepped, don't leave it without finishing (even if you restore the curtain) or some key assignments may be lost.

FURTHER ASSISTANCE ON LF

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

NOTES

TECHNICAL DETAILS

XROM: 10,05 | **LF** | SIZE: 000

Stack Usage:
- 0 T: used
- 1 Z: 0
- 2 Y: temporary c
- 3 X: result
- 4 L: 177

Alpha Register Usage:
- 5 M:
- 6 N: ALL USED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED BUT RESTORED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R06:
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Flag Usage: ONLY FLAG 10 IS ALTERED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10: CLEARED
- 25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
E?	
OM	2D

Local Labels In This Routine:
00
14

Execution Time: 8.3 seconds.
(For 16 assignment registers)

Peripherals Required: NONE

- Interruptible? YES
- Execute Anytime? YES
- Program File: LF
- Bytes In RAM: 79
- Registers To Copy: 59

Other Comments:

APPENDIX G CONTINUED FROM PAGE 233.

PSUEDO XROM - An XROM number seen when a synthetic key assignment doesn't have an internal ROM function. For example, assigning BEEP (134) as 0,134 and 134,0 will show BEEP in the first case and XROM 24,00 in the second case. Both work. XROM 24,00 is a Psuedo XROM.

Q

Q-LOADER - A PPC term describing a class of synthetic key assignments or barcodes that "loads" the Q register (in right to left order) into program memory at the location of the program pointer. A Q-Loader may be assigned by using MK with inputs of 4, m, k where m is 16 thru 28, and k is the key code. Reference is PPC CJ, V7N6P38c and V7N7P21b.

Q REGISTER - The tenth register (009 absolute) of HP-41 memory. This register is used for temporary ALPHA scratch. Q must be used with great care for general scratch and non-normalized STO/RCL, because of frequent use by the micro-processor.

R

RAM - Random Access Memory. Poorly chosen nomenclature for a memory that can store information written into it by the user. A better nomenclature would be--write memory, since ROM's may also be random access. (See ROM herein.) The term "random access" originated in the early days of computing when most memories were sequential access (mercury delay lines, drums, magnetic tape).

RECALL b - The contents of status register b is placed (without normalization) into stack register X. The stack is lifted if the lift is enabled. Also see b-Register.

RELATIVE ADDRESS - The location of desired data with respect to the user's reference (i.e., the curtain). The data pointed to by the user's reference has relative address zero. If the user's reference is register zero, the T register, the relative address equals the absolute address.

ROM - Read Only Memory. A memory whose contents are determined in the manufacturing process, and whose contents can not be modified by the user. (See RAM herein.)

RUN MODE - A "NON-PRGM" mode that is implied, but not actually described by HP. A running program is not what is described by the run mode.

S

S.A.S.E. - Self Addressed Stamped Envelope used to request information from a volunteer organization. Members who can't get U.S. stamps should send a mailing label with Postal Coupons to cover the postage. 1 oz. air mail is three C 22 coupons.

SDS - HP Software Development System. The minimum system consists of a HP-41C system (41C, Card Reader and Printer), a ROM Simulator, an HP-85A, and system software. The ROM simulator plugs into the HP-85A and one HP-41 port. When loaded with your programs it acts like a ROM. When your software is debugged, the HP-85A extracts it from the simulator, processes it, and produces a tape or disc that is delivered to HP for ROM production.

SPECIAL DISPLAY CHARACTERS - The characters that appear in the display when certain bytes are viewed. For example, the "full man" - ⾡ , the "boxed star" - ⊠, the "mu" - μ, and the "underscore" _. XEQ BLDSPEC with Y = 0, X = N where N = 0, 1, 2, 3, 5, 6, 12,33, 34, 35, 38, 39, 40, 41, 59, 64, 91, 92, 93, 95, 96, or 127 to display the 22 special characters. The HP-41 displays 83 different characters; 59 standard, 22 special, and 2 geese.

STAR BURST - See boxed Star.

STATUS REGISTERS - Sixteen registers (00 thru 15 absolute) used by the HP-41 as operating scratch. Five registers - XYZTL-- are available for general use. The remaining registers are usable with synthetic instructions. Also see Memory Void. The Status registers display as: T, Z, Y, X, L, M, N, O, P, Q, ⊢, a, b, c, d, e. Six characters print differently from their display. In order: [, \,], ↑, _, ᵀ.

STORE b - A synthetic instruction that places the contents of register X into status register b (address pointer). The byte code is 145, 124. The X register contents are stored (fortunately) without normalization. See b Register herein.

STORE c - The contents of register X are placed without normalization into the c (curtain control) register. See c Register herein.

SWITCHED QUAD - The 82170A QUAD Memory module has adequate internal space to add a small IC. type switch (S.P.S.T.) to control the PWO line and place the module into deep sleep. Such a QUAD may be effectively switched on and off the bus.

SYNTHETIC INSTRUCTION - An instruction used in HP-41 programming that is not covered in HP's Owner's Handbook and cannot be entered by normal key strokes. Synthetic instructions are used in synthetic programming. Typical instructions are: STO b, ISG M, TONE J, RCL A, SCI IND ᵀ.

SYNTHETIC KEY ASSIGNMENT - Creation of non-standard key assignment by storing unusual bytes in the key assignment registers. **MK** does this automatically.

SYNTHETIC LABEL - A label such as LBLᵀA is a Global Label, LBL A is a Local Label.

SYNTHETIC PROGRAMMING - Programming that involves the use of synthetic instructions. General use of the term may apply to the study of, creation of, and use of synthetic instructions. Synthetic Programming was created, defined, and developedby PPC. Also see Synthetic Instruction.

SYNTHETIC TONE - A tone instruction that is created synthetically. The ten normal tones of the HP-41 are made up of TONE (Byte 159 decimal) and 0 thru 9 (Byte 0 thru 9). Synthetic tones append bytes 10 thru 127 to create a total of 108 new tones with six new frequencies with durations of .023 to 5 seconds.

T

T REGISTER - The first register (000 absolute) of HP-41 memory. The T register is the "top" register of the four register (RPN) Stack--XYZT. Also see Status Registers.

APPENDIX G CONTINUED ON PAGE 273.

LG - PPC LOGO

This routine will add a special-graphics representation of the PPC logo to the current contents of the 41C print buffer. The logo consists of 21 columns of graphics, filling roughly half of the capacity of the buffer.

Example 1. Create the printed line 'PPC MEMBER 5000'. Create it all single width and upper case.

Keystrokes	Display	Result
CF 12, CF 13		Upper case, Single width
XEQ LG	(Orig. X reg.)	Add logo to print buffer
ALPHA (space) MEMBER (space) 5000 ALPHA	Text in ALPHA	Places text into ALPHA
ACA	(X)	Adds text to print buffer
PRBUF	(X)	Prints buffer

```
PPC MEMBER 5000
```

Example 2. Create the line 'member PPC'. Make the logo double width, with text lower case.

Keystrokes	Display	Result
SF 13	(Orig. X reg.)	Set lower case condition
ALPHA MEMBER (space) ALPHA	Text in ALPHA	Places text in ALPHA
ACA	(X)	Adds text to print buffer
SF 12	(X)	Set double width condition
XEQ LG	(X)	Adds logo to print buffer
PRBUF	(X)	Prints buffer

```
member PPC
```

COMPLETE INSTRUCTIONS FOR LG

LG adds a 21-column special graphic version of the PPC logo to the 82143A print buffer. It may be added in either single or double width format (based on the status of flag 12). This routine works exclusively in the ALPHA register leaving the stack unchanged. Two synthetic text lines of length 14 and 7 characters respectively, eliminate the necessity for separate ACCOL or BLDSPEC instructions for each special column of thermal printer dots. At the point where the logo is desired, set the status of flag 12 and simply press XEQ LG . The routine works the same in a program as it does from the keyboard. The only restriction is that the print buffer overflow if it already is more than half full (more than 21 bytes). This will cause the logo to be split between two printed lines, as in example 3 below:

MORE EXAMPLES OF LG

Example 3. Create the line 'HE IS AN ENTHUSIASTIC PPC MEMBER'.

Keystrokes	Display	Result
CF 12, CF 13		Upper case, Single width
ALPHA HE IS AN ENTHUSIASTIC (space) ALPHA	Text in ALPHA	Places text in ALPHA
ACA	(X)	Adds text to buffer
XEQ LG	(X)	Adds logo to buffer
ALPHA (space) MEMBER ALPHA	Text	Places text in ALPHA
ACA	(X)	Adds text to buffer
PRBUF	(X)	Prints buffer

```
HE IS AN ENTHUSIASTIC PP
C MEMBER
```

Routine Listing For: LG	
01♦LBL "LG"	07 ACSPEC
02 "Ω|<zΩf>£=x"	08 X<> [
03 "⊢Ω≠ N"	09 ACSPEC
04 X<>]	10 X<>]
05 ACSPEC	11 RTN
06 X<> \	

The synthetic text lines are as follows:

Line 02 (ALPHA 14 characters):
FE 11 C2 E4 7C 3C 7A F1 11 66 3E 1E 3D 78 F9

Line 03 (ALPHA append 7 characters):
F8 7F 11 9E 1D 9B BF 4E 87

LINE BY LINE ANALYSIS OF LG

Lines 02 and 03 place 21 characters into the ALPHA register; completely filling M, N and O.

Lines 04, 06, 08 and 10 preserve the contents of X by exchanging contents with M, N and O.

Lines 05, 07 and 09 add 7 printer columns of information at a time to the print buffer.

FURTHER DISCUSSION OF LG

Restrictions with LG :

There is a limited amount of additional information that can be added to the print buffer along with the logo, based on whether the logo has been accumulated in single or double width mode. In addition, the width mode of the characters added will affect the number of posible additions. Table 1 below describes these limitations.

Logo Width	Character Width	Maximum Number of Additional Char's
Standard	Standard	20
Double	Standard	18
Standard	Double	10
Double	Double	9

Table 1. Maximum number of ALPHA characters possible on a printed line in addition to the PPC logo (generated by LG).

Compatibility With Other Peripheral Routines:
The LG Routine is compatible with any of the other PPC ROM peripheral routines, since it does not use any numeric data registers or flags. Be sure, however, that there is no important information in the ALPHA register when LG is called, since it destroys the previous contents of the M, N and O registers.

REFERENCES FOR LG

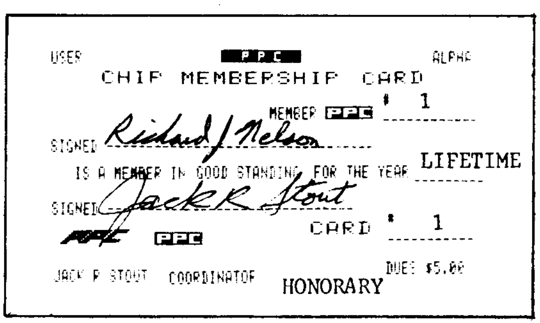
See PPC Calculator Journal V7N10P13a

CONTRIBUTORS HISTORY FOR LG

The original idea was put forth by Richard Nelson (1), and various logo shapes were tried. Inputs came from David Zarum (4736), Jake Schwartz (1820) and Roger Hill (4940), and adoption of one of Roger's logos was finally agreed upon. The actual logo chosen represents a compromise between a more clear version which would occupy more print buffer positions, and a logo with vertical sides which takes fewer positions. Since the actual logo has slanted sides and the print buffer is so limited, the logo presented here was selected for inclusion in the PPC ROM .

FINAL REMARKS FOR LG

Jack Stout (1221) used three experimental versions of the PPC logo in designing a membership card for members of the CHIP (Chicago Area PPC) chapter. The card appears below:


USER PPC ALPHA
CHIP MEMBERSHIP CARD
MEMBER PPC # 1
SIGNED Richard J Nelson
IS A MEMBER IN GOOD STANDING FOR THE YEAR LIFETIME
SIGNED Jack R Stout
CARD # 1
PPC
JACK R STOUT COORDINATOR HONORARY DUE: $5.00

FURTHER ASSISTANCE ON LG

Contact Jake Schwartz (1820) at 7700 Fairfield St., Phila., Penna. 19152 (home phone 215-331-5324); or Roger Hill (4940) at 300 S. Main St., Apt 5, Edwardsville, Ill. 62025 (home phone 618-656-8825).

T E C H N I C A L   D E T A I L S

XROM: 20,24 | LG | SIZE: 000

Stack Usage:
0 T:
1 Z: NONE USED
2 Y:
3 X:
4 L:

Alpha Register Usage:
5 M: USED
6 N: USED
7 O: USED
8 P:

Other Status Registers:
9 Q:
10 ⊦: NONE USED
11 a:
12 b:
13 c:
14 d:
15 e:

ΣREG: NOT USED.

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: NONE
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: ANY

Angular Mode: NOT USED

Unused Subroutine Levels: 5

Global Labels Called:
Direct: NONE
Secondary: NONE

Local Labels In This Routine:
NONE

Execution Time: Approx. 1 second.

Peripherals Required: 82143A Printer

Interruptible? YES
Execute Anytime? YES
Program File: LG
Bytes In RAM: 45
Registers To Copy: 29

Other Comments:
Uses roughly half of the print buffer (21 bytes).

LR - LENGTHEN RETURN STACK

The 41C operating system provides for six levels of subroutine calls by storing the six return addresses in status registers a and b. If more pending returns are needed, existing returns can be stored in a data register pair by the LR routine, freeing the status registers to held six more addresses. However, there can be a maximum of five return addresses pending when LR is called, since the instruction XEQ LR uses the sixth return address. The old return addresses can be restored by the SR routine.

Example 1: Suppose that you wish to call a hypothetical ROM routine XX, which is known to use three subroutine levels from the fourth subroutine level of your program ABC. This requires seven subroutine levels and normally would not be possible on the 41C. However, by using a single call to LR (and to SR) up to eleven levels may be used. The following program uses registers 11 and 12 to store the return stack.

```
LBL "ABC"
   :
LBL 01     LBL 01 called at 4th subroutine level
   .
11
XROM LR    Saves return stack in registers 11 & 12
XROM XX    Can freely use up to six subroutine levels
11
XROM SR    Restores original return stack
   :
END
```

COMPLETE INSTRUCTIONS FOR LR

LR will store up to five subroutine return addresses in a data register pair selected by the user. The routine does not alter the contents of status registers a or b. The user's program must put the number of the first register of the pair in the X register before LR is called. The Y, Z, and T registers are returned in X, Y, and Z after execution, and LastX and all ALPHA registers are lost.

SR recalls five return addresses from a data register pair and stores them in status registers a and b. The information in the data register pair is not altered by SR, and may be used again if desired. The number of the first register of the pair must be in X when SR is called, and Y, Z, and T are returned in X, Y, and Z after execution. LastX and ALPHA are lost.

MORE EXAMPLES OF LR

Example 2: The SUB1 routine is a demonstration of extended subroutine stack depth. The user places in X the desired subroutine depth, which can be up to 770 levels, depending on the amount of available memory. The formula is max levels = 5*[INT (SIZE/2) +1]. When the routine is run, it executes repeated subroutine calls, displaying the current subroutine level, until the desired depth has been reached, when it beeps and starts executing repeated returns, counting back down until all subroutines have been returned from. This program was written by Keith Jarett (4360) as a test routine for LR and SR during the ROM loading process.

BAR CODE ON PAGE 483

APPLICATION PROGRAM FOR: LR

```
01♦LBL "SUB1"      31♦LBL 14
02 E3              32 R↑
03 /               33 R↑
04♦LBL 01          34 XEQ 01
05 VIEW X          35 RCL X
06 ISG X           36 E
07 GTO 14          37 X=Y?
08 BEEP            38 GTO 14
09 INT             39 -
10 DSE X           40 5
11 RTN             41 XROM "QR"
12♦LBL 14          42 X≠0?
13 RCL X           43 GTO 14
14 INT             44 RDN
15 E               45 E
16 X=Y?            46 -
17 GTO 14          47 2
18 -               48 *
19 5               49 XROM "SR"
20 XROM "QR"       50 R↑
21 X≠0?            51 R↑
22 GTO 14          52♦LBL 14
23 RDN             53 R↑
24 E               54 R↑
25 -               55 VIEW X
26 2               56 DSE X
27 *               57 RTN
28 XROM "LR"       58 PSE
29 R↑              59 VIEW X
30 R↑              60 BEEP
                   61 .END.
```

APPLICATION PROGRAM 1 FOR

LR and SR are simple to use when the depth of subroutine calls is a constant. However, for recursive algorithms the program determines the depth of subroutine usage, and managing the return stack becomes more difficult. The two programs LRR (lengthen return stack for recursion) and SRR (shorten return stack for recursion) provide automatic management of the return stack by calling LR and SR only when needed. These routines require two data registers for level counting, two registers for each use of LR, and a short initialization (IRX) before the routines can be used. LRR and SRR automatically allow for curtain moving, which is usually needed to support recursion. They use the top 2+2k data registers, where k represents the maximum number of times LR is called. Since LR is called for each 5 subroutine levels this means that 2+2*INT(n/5) registers are used, where n is the maximum number of subroutine levels. The top data register is used by LRR and SRR as a subroutine level counter, while the penultimate register contains a pointer of the form iii.fff02 used to access registers for LR and SR, with iii ≥ fff.

The return stack management routines are used as follows:

1) Make sure your SIZE is sufficient for what you want to do. Then place in X the number of lowest register to be used for the extended return stack. (The return stack is actually constructed from high registers to low registers, but this number will provide a lower bound to protect other data that you may need. If you don't need this protection use 1 or 0.) XEQ "IRX" (initialize for recursive execution) to initialize the top two registers for LRR and SRR.

2) After each LBL which initiates a chain of calls more than two deep (this includes all recursive labels, but does not include utility routines which themselves call only one level) you must XEQ "LRR". The XEQ "LRR" may be anywhere between the LBL and the first XEQ instruction, but the recommended location is immediately after the LBL.

3) Likewise, before a RTN is executed from a program segment that initiates a chain of calls more than two deep you must XEQ "SRR". The XEQ "SRR" may be anywhere between the last XEQ instruction and the RTN, but the recommended location is immediately before the RTN. It is also recommended that all return paths be funnelled to a single RTN instruction , so that only one XEQ "SRR" is required.

4) Because of the nature of LRR and SRR it is always possible to call a two level subroutine without using LRR and SRR. In this way, utility routines which themselves call at most one other subroutine may be used efficiently. An example of this technique is the use of PUSH and POP in Example 4.

5) Only two parameters in the stack (X and Y registers) remain intact after execution of LRR or SRR. However, you may insert any number of steps between the LRR call and its associated RTN. In this way, the stack and ALPHA may be emptied or filled as required.

6) The routines are shown here with global labels for clarity; however, if possible they should be used with local labels to allow the XEQ branches to be compiled. This will increase execution speed as well as reduce the byte count.

Example 3: The SUB2 routine operates identically to the SUB1 routine, except for some unavoidable display scrolling (see IF Example 6), but it has been modified to use LRR and SRR. The modified version is more compact and is easier to understand, since return stack management is not performed by the routine itself. However, the SUB1 routine is more efficient because it does not use data registers (all indexing is done in the stack) and it is shorter because it only calls PPC ROM routines, whereas LRR, SRR, and IRX must all be in memory for SUB2 to operate. There is an interesting tradeoff, though, because a routine such as SUB2 can be written and debugged in a much shorter time. Unless the maximum capacity of the 41C is needed, it is probably not worth the required programming effort to make your routine perform its own return stack management.

BAR CODE ON PAGE 484

APPLICATION PROGRAM FOR: **LR**

```
01◆LBL "SUB2"
02 E3
03 /
04 E
05 XEQ "IRX"

06◆LBL 01
07 XEQ "LRR"
08 VIEW X
09 ISG X
10 GTO 02
11 TONE 9
12 INT
13 GTO 03

14◆LBL 02
15 XEQ 01
16 VIEW X

17◆LBL 03
18 XEQ "SRR"
19 DSE X
20 RTN
21 PSE
22 TONE 5
23 CLD
24 END
```

BAR CODE ON PAGE 484

APPLICATION PROGRAM FOR: **LR**

```
01◆LBL "IRX"
02 CHS
03 .02
04 +
05 XROM "S?"
06 E
07 -
08 .
09 STO IND Y
10 RDN
11 +
12 E3
13 ST/ Y
14 RDN
15 DSE L
16 STO IND L
17 RDN
18 RTN

19◆LBL "LRR"
20 SF 14
21 GTO 00

22◆LBL "SRR"
23 CF 14

24◆LBL 00
25 RCL c
26 STO [
27 "├◆◆◆◆"
28 X<> [
29 X<> d
30 CF 02
31 CF 03
32 X<> d
33 ENTER↑
34 INT
35 HMS
36 X<>Y
37 SIGN
38 RDN
39 7
40 ST* Y
41 X<> L
42 +
43 - E1
44 *
45 INT
46 64
47 MOD
48 SF 25

49◆LBL 01
50 RCL IND X
51 FC? 25
52 GTO 02
53 X<> L
54 +
55 GTO 01

56◆LBL 02
57 E
58 -
59 STO [
60 X<> L
61 FC?C 14
62 GTO 05
63 X<> IND L
64 ST+ IND L
65 X=0?
66 GTO 04
67 5
68 MOD
69 X≠0?
70 GTO 04
71 DSE [
72 RDN
73 ISG IND [
74 FS? 53
75 GTO 03
76 RCL IND [
77 INT
78 ST- [
79 X<> [
80 GTO "LR"

81◆LBL 03
82 "NO ROOM- LRR"
83 PROMPT

84◆LBL 04
85 RDN
86 CLA
87 RTN

88◆LBL 05
89 ST- IND L
90 RDN
91 RCL IND L
92 X=0?
93 GTO 04
94 5
95 MOD
96 X≠0?
97 GTO 04
98 DSE [
99 RDN
100 RCL IND [
101 INT
102 X=0?
103 GTO 06
104 DSE IND [
105 ""
106 ST- [
107 X<> [
108 GTO "SR"

109◆LBL 06
110 "TOO FAR-SRR"
111 PROMPT
112 END
```

The LRR and SRR routines are especially useful for implementing recursive algorithms on the 41C. If your algorithm is not recursive, the direct method of Example 1 may be better. There are many problems that lend themselves to recursive solutions--one of the simplest of these is the computation of a factorial. With a high level computer language that supports recursion, a factorial algorithm could be implemented in the following two statements:

```
PROCEDURE FACT(N)
FACT    = 1
IF N = 1 THEN RETURN
ELSE FACT    = N * FACT(N-1)
```

If the 41C had a large enough operational stack and return stack, a factorial routine could be written as follows:

```
01 LBL "FCT"
02 ENTER↑
03 DSE X
04 XEQ "FCT"
05 X=0?
06 SIGN
07 *
08 RTN
```

The zero test and SIGN merely prevent multiplication by zero. This routine correctly calculates the factorial of 1, 2, or 3 but fails for larger numbers, because all four stack registers are used. By using the PUSH and POP routines (and the IRX initialization routine) to form an indefinitely long stack in memory, and the LRR and SRR routines to provide an extended return stack, a recursive factorial routine can easily be written for the HP-41.

Example 4: This program is given for illustrative purposes only. Because of its simplicity, it is a good example of recursive programming techniques; however, it obviously has no use as a computational tool since the 41C FACT function is hundreds of times faster and uses no data registers.

Both the return stack management routines and the 'infinite' stack routines need initialization before FCT can be run. To evaluate up through 69!, execute the following:

SIZE $\geq$ 98 (= n + 3 + 2 * INT (n/5))

XEQ "IRX"

To evaluate factorials, just enter an integer between 1 and 69 and XEQ "FCT". The registers do not have to be re-initialized unless the program is stopped before completion or if register 00 or the highest two data registers are altered.

APPLICATION PROGRAM FOR: **LR**

```
01♦LBL "FCT"        16♦LBL "PUSH"
02 XEQ "LRR"        17 STO IND 00
03 XEQ "PUSH"       18 ISG 00
04 DSE X            19 ""
05 GTO 21           20 RTN
06 X=0?
07 SIGN             21♦LBL "POP"
08 GTO 22           22 DSE 00
                    23 ""
09♦LBL 21           24 RCL IND 00
10 XEQ "FCT"        25 RTN

11♦LBL 22           26♦LBL "INIT"
12 XEQ "POP"        27 E
13 *                28 STO 00
14 XEQ "SRR"        29 XEQ "IRX"
15 RTN              30 END
```

While the factorial program has a simpler non-recursive solution on the 41C, there are many routines that are extremely difficult to solve unless recursive methods are used. An example of this is the Towers of Hanoi program by Harry Bertuccelli (3994), covered elsewhere in this manual.

APPLICATION PROGRAM 2 FOR LR

If your recursive program calls itself from only one point, then the return addresses stored by LR are redundant. This means that there is probably a simple nonrecursive looping solution to your problem, but if you want to use recursion you need not pay LRR's heavy penalty in register usage.

The LRS (lengthen return stack with single return address) and SRS (shorten return stack with single return address) supportive routines are similar to LRR and SRR with the following exceptions. LR is called only once, at the fifth level. LRS assumes that all return addresses are identical. SRS calls SR every five levels as does SRR, but it always places the same return pointers in status registers a and b. Only the top three data registers are used. No input is required for ISX (initialize for single return address execution).

BAR CODE ON PAGE 484

APPLICATION PROGRAM FOR: **LR**

```
01♦LBL "ISX"        46 GTO 01
02 XROM "S?"
03 DSE X            47♦LBL 02
04 .                48 E
05 STO IND Y        49 -
06 R↑               50 FC?C 14
07 R↑               51 GTO 04
08 RTN              52 X<> L
                    53 X<> IND L
09♦LBL "LRS"        54 ST+ IND L
10 SF 14            55 5
11 GTO 00           56 X=Y?
                    57 GTO 03
12♦LBL "SRS"        58 R↑
13 CF 14            59 R↑
                    60 RTN
14♦LBL 00
15 RCL c            61♦LBL 03
16 STO [            62 R↑
17 "⊢♦♦♦♦"          63 R↑
18 X<> [            64 LASTX
19 X<> d            65 2
20 CF 02            66 -
21 CF 03            67 GTO "LR"
22 X<> d
23 ENTER↑           68♦LBL 04
24 INT              69 STO [
25 HMS              70 X<> L
26 X<>Y             71 ST- IND L
27 SIGN             72 RDN
28 RDN              73 RCL IND L
29 7                74 X=0?
30 ST* Y            75 STO IND L
31 X<> L            76 X=0?
32 +                77 GTO 05
33 - E1             78 5
34 *                79 MOD
35 INT              80 X≠0?
36 64               81 GTO 05
37 MOD              82 X<> [
38 SF 25            83 CLA
39 CLA              84 2
                    85 -
40♦LBL 01           86 GTO "SR"
41 RCL IND X
42 FC? 25           87♦LBL 05
43 GTO 02           88 RDN
44 X<> L            89 END
45 +
```

APPLICATION PROGRAM FOR:	LR

```
01♦LBL "SUB2"          12♦LBL 02
 02  E3                13 XEQ "LRS"
 03 /                   14 XEQ 01
 04 XEQ "ISX"           15 XEQ "SRS"
                        16 VIEW X
 05♦LBL 01
 06 VIEW X              17♦LBL 03
 07 ISG X               18 DSE X
 08 GTO 02              19 RTN
 09 TONE 9              20 PSE
 10 INT                 21 TONE 5
 11 GTO 03              22 CLD
                        23 END
```

The modified versions of SUB2 and FCT shown here use LRS and SRS. This saves registers and increases speed. Both SUB2 and FCT satisfy the essential constraint that there is only one XEQ instruction that is recursive. Because of this constraint it is simplest to surround the recursive XEQ instruction with XEQ "LRS" above and XEQ "SRS" below.

BAR CODE ON PAGE 484

APPLICATION PROGRAM FOR:	LR

```
 01♦LBL "FCT"           16♦LBL "PUSH"
 02 XEQ "PUSH"          17 STO IND 00
 03 DSE X                18 ISG 00
 04 GTO 21               19 ""
 05 X=0?                 20 RTN
 06 SIGN
 07 GTO 22              21♦LBL "POP"
                         22 DSE 00
 08♦LBL 21               23 ""
 09 XEQ "LRS"           24 RCL IND 00
 10 XEQ "FCT"            25 RTN
 11 XEQ "SRS"
                        26♦LBL "INIT"
 12♦LBL 22               27  E
 13 XEQ "POP"            28 STO 00
 14 *                    29 XEQ "ISX"
 15 RTN                  30 END
```

LINE BY LINE ANALYSIS OF LR

Status registers a and b contain the program pointer and six return addresses - each of these are two bytes (16) long. In the following analysis, a letter "P" will be used to represent each byte of the program pointer, a digit "1" for each byte of the first return address, a "2" for the second return address and so on. Using this notation, registers a and b have the following configuration:

3	2	2	1	1	P	P	b
6	6	5	5	4	4	3	a

To extend the return stack, only the second through the sixth return addresses must be stored- the first return address provides a return to the program that called LR or SR, and the pointer just contains the absolute address of the program step where register b was recalled. The five return addresses that are

STACK AND ALPHA REGISTER ANALYSIS FOR LR

T	Z	Y	X	L	P							O							N							M							L-#	INSTRUCTION
T_0	Z_0	Y_0	n	L																													26	♦LBL "LR"
T_0	Z_0	Y_0	1	n																													27	SIGN
1	T_0	Z_0	Y_0	n																													28	RDN
1	T_0	Z_0	Y_0	n																												+	29	"+"
T_0	Z_0	Y_0	6655443	n																												+	30	RCL a
T_0	Z_0	Y_0	6655443	n															6	6	5	5	4	4	3							+	31	STO \
6655443	T_0	Z_0	Y_0	n															6	6	5	5	4	4	3							+	32	RDN
T_0	Z_0	Y_0	32211PP	n															6	6	5	5	4	4	3							+	33	RCL b
T_0	Z_0	Y_0	+	n															6	6	5	5	4	4	3	3	2	2	1	1	P	P	34	X<> [
T_0	Z_0	Y_0	+	n														+	6	6	5	5	4	4	3	3	2	2	1	1	P	P	35	STO]
T_0	Z_0	Y_0	+	n														+	6	6	5	5	4	4	3	3	2	2	1	1	P	P	36	ASTO IND L
T_0	Z_0	Y_0	+	n+1														+	6	6	5	5	4	4	3	3	2	2	1	1	P	P	37	ISG L
T_0	Z_0	Y_0	+	n+1														+	6	6	5	5	4	4	3	3	2	2	1	1	P	P	38	""
T_0	Z_0	Y_0	+	n+1									+	6	6	5	5	4	4	3	3	2	2	1	1	P	P	*	*	*	*	*	39	"├*****"
T_0	Z_0	Y_0	+	n+1														+	4	3	3	2	2	1	1	P	P	*	*	*	*	*	40	STO]
T_0	Z_0	Y_0	+	n+1														+	4	3	3	2	2	1	1	P	P	*	*	*	*	*	41	ASTO IND L
+	T_0	Z_0	Y_0	n+1														+	4	3	3	2	2	1	1	P	P	*	*	*	*	*	42	RDN
+	T_0	Z_0	Y_0	n+1																													43	CLA
+	T_0	Z_0	Y_0	n+1																													44	RTN

Routine Listing For:	LR

```
26♦LBL "LR"          36 ASTO IND L
27 SIGN               37 ISG L
28 RDN                38 ""
29 "+"                39 "├*****"
30 RCL a              40 STO ]
31 STO \              41 ASTO IND L
32 RDN                42 RDN
33 RCL b              43 CLA
34 X<> [             44 RTN
35 STO ]
```

stored by LR constitute ten bytes, five of which are stored in each register of the pair. Since some of the bytes may be nulls (hex 00), and they are stored directly from the ALPHA register using ASTO (which will store six characters, but skips over leading nulls), an alpha character delimiter must be used to force ASTO to take the correct five bytes. This delimiter is the character "+", which is the sixth byte stored in each register. After execution of LR, the register pair contains the following information stored as alpha strings:

Reg n : "+66554"

Reg n+ 1 : "+43322"

The SR routine expects to find the return addresses store in this form and merges these addresses onto the current program pointer and first return address and then stores the results into registers a and b.

The majority of both routines consists of ALPHA register shifting--to analyze this in detail, it is probably easiest to use a STACK/ALPHA analysis sheet such as the one in *PPC TECHNICAL NOTES*, V1N3P38. Analysis sheets for LR and SR are printed in this manual. A blank analysis sheet may be found on page 259 of the manual.

CONTRIBUTORS HISTORY FOR LR

Harry Bertuccelli (3994) wrote the first subroutine level extension routines (see *PPC CALCULATOR JOURNAL*, V7N6P8). Paul Lind (6157) completely rewrote LR and SR, and Roger Hill (4940) independently wrote virtually identical routines.

The application programs were written by Harry Bertuccelli and Keith Jarett (4360) based on Paul Lind's idea.

FURTHER ASSISTANCE ON LR

Call Paul Lind (6157) at (206) 525-1033.
Call Harry Bertuccelli (3994) at (213) 846-6390.

NOTES

TECHNICAL DETAILS

XROM: 20,02	LR	SIZE: 002

Stack Usage:	**Flag Usage:** NONE USED
0 T: USED	04:
1 Z: PREVIOUS T	05:
2 Y: PREVIOUS Z	06:
3 X: PREVIOUS Y	07:
4 L: X + 1	08:
Alpha Register Usage:	09:
5 M:	10:
6 N: ALL CLEARED	
7 O:	
8 P:	25:
Other Status Registers:	**Display Mode:** UNCHANGED
9 Q: NOT USED	
10 ⊢: NOT USED	
11 a: NOT USED	**Angular Mode:** UNCHANGED
12 b: NOT USED	
13 c: NOT USED	
14 d: NOT USED	**Unused Subroutine Levels:** 5 CALLED BY A PROGRAM
15 e: NOT USED	6 FROM THE KEYBOARD
ΣREG: UNCHANGED	**Global Labels Called:**
Data Registers:	**Direct**: NONE — **Secondary**: NONE
R00: TWO CONSECUTIVE REGISTERS SPECIFIED BY THE USER ARE	
R06: ALTERED	
R07:	
R08:	**Local Labels In This Routine:** NONE
Execution Time: .7 seconds.	
Peripherals Required: NONE	
Interruptible? YES	**Other Comments:**
Execute Anytime? NO	
Program File: SR	
Bytes In RAM: 40	
Registers To Copy: 40	

STACK AND ALPHA REGISTER ANALYSIS FOR ______________

PPC

T	Z	Y	X	L	P	O	N	M	L-#	INSTRUCTION

PPC

M1 - MATRIX, INTERCHANGE ROWS

M1 is one of five matrix routines. M1 will interchange two rows in a matrix. The normal input to M1 is simply the numbers of the two rows to be interchanged. M1 may also be considered part of the data base management routines IR and DR. M1 can be used to interchange two records in a file. Input in this case is the two record numbers.

In addition, each of the five matrix routines requires two stored values, one of which is the starting register of the matrix and the other is the number of columns in the matrix. Matrices are assumed to be stored with each row occupying a consecutive block of registers. Thus the number of columns is the block size and the entire matrix is stored row by row as one string of consecutive registers. R07 holds the starting register of the matrix and R08 holds the number of columns.

Example 1: Use M1 to interchange rows 2 and 4 in the following 6x5 matrix which is assumed to be stored in registers R15-R44.

21	35	55	74	83
11	93	56	36	29
65	78	32	27	75
53	94	46	62	97
54	39	61	67	82
23	45	77	15	25

For this first example we will explicitly show the correspondence between the data registers and the matrix elements. The element in the upper left-hand corner is assumed to be in row 1 and column 1. Store the matrix entries in the following registers. You may wish to record the results on a magnetic card as other ROM routine write-ups use this matrix as an example.

R15: 21	R23: 36	R31: 94	R39: 82
R16: 35	R24: 29	R32: 46	R40: 23
R17: 55	R25: 65	R33: 62	R41: 45
R18: 74	R26: 78	R34: 97	R42: 77
R19: 83	R27: 32	R35: 54	R43: 15
R20: 11	R28: 27	R36: 39	R44: 25
R21: 93	R29: 75	R37: 61	
R22: 56	R30: 53	R38: 67	

Since this matrix starts in R15 and the number of columns in the matrix is 5 we must first store the following in R07 and R08.

R07: 15=starting register   R08: 5=# of columns

Any number of matrix operations may be performed on the above matrix without changing the numbers in R07 and R08. (These operations include M1, M2, M3, M4, or M5)

Now to interchange rows 2 and 4 key 2 ENTER↑ 4 (the order of the two row numbers is unimportant) and XEQ "M1". The above matrix will then change to the following:

21	35	55	74	83
53	94	46	62	97
65	78	32	27	75
11	93	56	36	29
54	39	61	67	82
23	45	77	15	25

The storage in the data registers is now:

R15: 21	R23: 62	R31: 93	R39: 82
R16: 35	R24: 97	R32: 56	R40: 23
R17: 55	R25: 65	R33: 36	R41: 45
R18: 74	R26: 78	R34: 29	R42: 77
R19: 83	R27: 32	R35: 54	R43: 15
R20: 53	R28: 27	R36: 39	R44: 25
R21: 94	R29: 75	R37: 61	
R22: 46	R30: 11	R38: 67	

COMPLETE INSTRUCTIONS FOR M1

1) The matrix is assumed to be stored with each row occupying a consecutive block of registers. The entire matrix is assumed to be stored row by row as one string of consecutive data registers.

2) The number of the starting register for the matrix is to be stored in R07. The number of columns in the matrix is to be stored in R08. Both the row and column numbers start counting from 1.

3) To interchange any two rows in the matrix key in the two row numbers in Y and X (the order is unimportant) and XEQ "M1". M1 performs a block exchange of the two rows involved by dropping into the routine BE.

MORE EXAMPLES OF M1

Example 2: Use M1 to interchange rows 3 and 5 in the following 5x5 matrix which is assumed to be stored in R20-R44.

13	19	17	14	18
25	31	42	15	23
6	34	87	92	14
12	9	24	36	49
17	14	13	12	18

The following shows the matrix elements stored in R20-R44.

R20: 13	R29: 23	R38: 36
R21: 19	R30: 6	R39: 49
R22: 17	R31: 34	R40: 17
R23: 14	R32: 87	R41: 14
R24: 18	R33: 92	R42: 13
R25: 25	R34: 14	R43: 12
R26: 31	R35: 12	R44: 18
R27: 42	R36: 9	
R28: 15	R37: 24	

Store the starting register for the matrix in R07. 20 STO 07. Store the number of columns in R08. 5 STO 08. Then to perform the row interchange between rows 3 and 5 key 3 ENTER↑ 5 and XEQ "M1".

The matrix will now appear as:

$$\begin{bmatrix} 13 & 19 & 17 & 14 & 18 \\ 25 & 31 & 42 & 15 & 23 \\ 17 & 14 & 13 & 12 & 18 \\ 12 & 9 & 24 & 36 & 49 \\ 6 & 34 & 87 & 92 & 14 \end{bmatrix}$$

and the following list of registers reflects this change.

R20: 13	R29: 23	R38: 36
R21: 19	R30: 17	R39: 49
R22: 17	R31: 14	R40: 6
R23: 14	R32: 13	R41: 34
R24: 18	R33: 12	R42: 87
R25: 25	R34: 18	R43: 92
R26: 31	R35: 12	R44: 14
R27: 42	R36: 9	
R28: 15	R37: 24	

Example 3: This example will illustrate the use of M1 to exchange records in a file. See Example 1 of the routine IR. The following list of records is used as an example file.

Record #1: Mary Adams
354-1662
Gary, IN

Record #2 Jane Hamilton
363-5648
Boston, MA

Record #3 Robert Jefferson
261-2347
Fresno, CA

Record #4 Mike Johnson
745-3254
Denver, CO

Record #5 James Masterson
565-2314
Toledo, OH

Record #6 Joe Robinson
756-4438
Peoria, IL

This sample file is stored in data registers R10-R45 where each record consists of 6 consecutive data registers.

R10: Mary	R28: Mike
R11: Adams	R29: Johnso
R12:	R30: n
R13: 354.1662	R31: 745.3254
R14: Gary	R32: Denver
R15: IN	R33: CO
R16: Jane	R34: James
R17: Hamilt	R35: Master
R18: on	R36: son
R19: 363.5648	R37: 565.2314
R20: Boston	R38: Toledo
R21: MA	R39: OH
R22: Robert	R40: Joe
R23: Jeffer	R41: Robins
R24: son	R42: on
R25: 261.2347	R43: 756.4438
R26: Fresno	R44: Peoria
R27: CA	R45: IL

The following information in registers R07, R08, and R09 is used for the purpose of identifying the file.

R07: starting register of entire file
R08: number of registers per record
R09: total number of records in the file

The information in R09 is not required by M1 but is required by IR and DR. The routines M1 - M5, IR, DR, UR, and PR are all designed to be compatible, even though they seem to perform totally unrelated functions. The storage of records in a file corresponds exactly to the storage of the row elements in a matrix.

The number in R07 is the starting register of the matrix or file, whichever is conceptually assumed to occupy the data registers. For this example store 10 in R07. The number in R08 is the number of columns in the matrix, or the number of registers per record, again, whichever is conceptually assumed to occupy the data registers. For this example store 6 in R08.

Now if we want to exchange the registers occupied by Jane Hamilton and Mike Johnson (records 2 and 4) simply key 2 ENTER↑ 4 and XEQ "M1". The records are now in the logical form:

Record #1: Mary Adams
354-1662
Gary, IN

Record #2 Mike Johnson
745-3254
Denver, CO

Record #3 Robert Jefferson
261-2347
Fresno, CA

Record #4 Jane Hamilton
363-5648
Boston, MA

Record #5 James Masterson
565-2314
Toledo, OH

Record #6 Joe Robinson
756-4438
Peoria, IL

and in the data registers as:

R10: Mary	R28: Jane
R11: Adams	R29: Hamilt
R12:	R30: on
R13: 354.1662	R31: 363.5648
R14: Gary	R32: Boston
R15: IN	R33: MA
R16: Mike	R34: James
R17: Johnso	R35: Master
R18: n	R36: son
R19: 745.3254	R37: 565.2314
R20: Denver	R38: Toledo
R21: CO	R39: OH

R22: Robert	R40: Joe
R23: Jeffer	R41: Robins
R24: son	R42: on
R25: 261.2347	R43: 756.4438
R26: Fresno	R44: Peoria
R27: CA	R45: IL

To further exchange Mary Adams and James Masterson (records 1 and 5) key 1 ENTER↑ 5 and XEQ "M1". Then check the data registers to see that the proper exchange has been made.

APPLICATION PROGRAM 1 FOR M1

Matrix Support Routines RRM AND MIO

The routines called RRM and MIO are provided as matrix support routines. RRM calls the ROM routines M1, M2, M3, M4, M5, and BX. MIO calls M4 and M5.

The program titled RRM will transform a matrix into row reduced echelon form. This means the program will calculate determinants and inverses and will solve systems of equations. The RRM program is only 70 lines long (104 bytes), and handles the three matrix problems, either individually or simultaneously, and uses the technique known as partial pivoting which helps reduce round-off error. Moreover, the only limitation on the size of the matrices is the number of available data registers. The RRM program can even be applied to more than one matrix in data memory. If more than 319 registers ever become available for the HP-41C the RRM program may be run without any modifications to handle any size matrix.

Given our present limitation of 319 registers RRM can be used to compute the determinant of a 17x17 matrix, to solve a system of up to 16 linear equations in 16 unknowns, or compute the inverse of a 12x12 matrix. To solve any of these problems simply load the appropriate matrix in the 41C and XEQ "RRM". The desired result, whether it be a determinant or an inverse or the solution to a system of equations will be calculated and left in the 41C.

The second program called MIO is to be used for matrix input/output operations that will automatically store or recall the entries of a matrix consistent with the requirements of the ROM matrix routines M1 - M5. Although RRM and MIO can be merged into one program, the reason for writing two separate matrix modules is to handle as large a matrix as possible. RRM does all the hard work; MIO is only an example of an input/output scheme.

It should be pointed out that it is possible to use other methods to solve the same matrix problems, but for completely automatic operation as RRM and MIO provide we have approached the theoretical limit. If you plan on writing your own matrix routines that will call M1 - M5 the following suggestions will be helpful.

1. M1 - M5 require the starting register of the matrix to be stored in R07 and the number of columns in the matrix be stored in R08. Matrices are stored row by row with each row occupying a block of consecutive registers. The entire matrix is stored as one large block of consecutive registers.

2. Although M1 - M5 do not require the number of rows, the number of rows, if used, should be stored in R09. Later you may find matrix uses for IR and DR which do use R09.

3. To achieve maximum size start storing the matrix entries in R10 on up and use registers R06 and below for program scratch area.

4. Given the above 3 constraints consider further the storage requirements. For the determinant problem, to store a 17x17 matrix requires 289 registers. For the systems of equations problem to hold a 16x16 matrix and one extra column for the constants requires 16x17= 272 registers. For the inverse, unless you are calculating the inverse in place, you will need to store two matrices, one being the original and the other being a form of the identity matrix. Since 12x12 times 2 = 144x2 = 288, you will need 288 registers for the inverse of a 12x12 matrix.

Thus you will need a maximum of 289 registers to solve all 3 types of problems. If you are using R00-R09 for the registers your program requires then 299 registers will already be accounted for before you even start to enter your program. Since 319 - 299 = 20, your program may use approximately 20x7 = 140 bytes.

Both RRM and MIO have been restricted to use less than 140 bytes; RRM is 104 bytes and MIO is 129 bytes. If you are not particular about the maximum capacity available you can combine these two programs and add many of your own bells and whistles to MIO and still have enough data memory available to perform some fancy operations on 10x10 matrices. If RRM is used alone you will have 295 registers available for matrix data. If MIO is used alone 291 data registers will be available for matrix data. When RRM and MIO are combined 276 registers are available.

Three examples follow which illustrate the use of RRM and MIO. To run the examples, first perform "MEMORY LOST" and then SIZE 031 (minimum). Read in the MIO program first and then GTO .. Next read in the RRM program and GTO .. again. Then key CAT 1 and immediately press R/S so that you are in the MIO program. Switching to USER mode makes the following functions available on keys A, B, C.

New Matrix	Review Matrix	Recall (Y,X)

Problem 1: Solve the system of equations:

$$-5X + 10Y + 15Z = 5$$

$$2X + Y + Z = 6$$

$$X + 3Y - 2Z = 13$$

Perform the following operations.

Press	Function	See in Display
A	Initialization for a new matrix	"START REG. ?"
10 R/S	Start storing matrix in R10 and above	"DIM: R?↑C?"
3 ENTER↑ 4 R/S	Key in dimension as 3 rows and 4 columns.	-TONE 9- "(1,1)=?"

We next enter one by one the entries of the coefficient matrix starting with the first row.

$$\left[\begin{array}{ccc|c} -5 & 10 & 15 & 5 \\ 2 & 1 & 1 & 6 \\ 1 & 3 & -2 & 13 \end{array}\right]$$

The program will sound TONE 9 when it is ready for the next entry and will prompt with "(row,column)=?" where row and column are numbers. Key in the next coefficient followed by R/S. For example, the display should still show "(1,1)=?" and the first row would be keyed in as:

See in Display	Press
"(1,1)=?"	5 CHS R/S
"(1,2)=?"	10 R/S
"(1,3)=?"	15 R/S
"(1,4)=?"	5 R/S
"(2,1)=?"	

Continue keying in the 2nd and 3rd rows. After keying in 13 and pressing R/S for the last (3,4) entry the program will sound BEEP to indicate you should be finished entering the data.

You may now verify the data input by pressing B. First however, store a number (say 4) in R05 for the number of decimal places to be displayed. Pressing B will automatically run through the entire matrix. If a printer is connected and turned on key B will give a printout of the entire matrix. If you prefer scientific notation change line 43 in the MIO program from FIX IND 05 to SCI IND 05. A BEEP will sound when the output is finished.

You may also inspect any particular element using key C. Key in the row and column numbers of the matrix element you wish to view and press C. For example, to verify that the (3,2) element is 3, key 3 ENTER↑ 2 and press C. You should first see "R19.0000" and then "(3,2)=3.0000". The indication here is that the (3,2) element is stored in register R19 and is equal to 3.

Note: If you make an incorrect entry during the automatic input phase simply continue entering elements as directed by the display. After all entries have been made you can use key C to make corrections, since pressing C tells you in which register you should manually store the element in question.

To solve the above system simply XEQ "RRM". This first example will run in about 34 seconds. When the program ends key CAT 1 R/S to insure you are in the MIO program and then press B in USER mode to display the final matrix which is:

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right]$$

The solution is X=2, Y=3, and Z=-1. The determinant of the square coefficient matrix is stored in R01. det. = 150.0000

<u>Problem 2:</u> Find the inverse of the matrix:

$$\left[\begin{array}{ccc} 2 & -3 & 1 \\ 3 & 2 & -1 \\ 5 & -2 & 1 \end{array}\right]$$

To use RRM to find the inverse of a square matrix we form the auxiliary matrix which consists of the original matrix augmented by an identity matrix of the same size. For this problem we will input the 3x6 matrix:

$$\left[\begin{array}{ccc|ccc} 2 & -3 & 1 & 1 & 0 & 0 \\ 3 & 2 & -1 & 0 & 1 & 0 \\ 5 & -2 & 1 & 0 & 0 & 1 \end{array}\right]$$

Press	Function	See in Display
A	Initialization for a new matrix	"START REG. ?"
10 R/S	Start storing matrix in R10 and beyond	"DIM: R?↑C?"
3 ENTER↑ 6 R/S	Key in size as 3 rows and 6 columns.	-TONE 9- "(1,1)=?"

Now continue as in the first example and enter the matrix starting with the first row. Then XEQ "RRM". The program will finish in about 45 seconds. Key CAT 1 R/S when the program finishes and press B to display the result:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 0.1250 & 0.1250 \\ 0 & 1 & 0 & -1 & -0.3750 & 0.6250 \\ 0 & 0 & 1 & -2 & -1.375 & 1.6250 \end{array}\right]$$

The right hand 3x3 matrix is the inverse of the original matrix. The determinant of the original matrix is found by recalling R01. det. = 8.0000

<u>Problem 3:</u> Use RRM to simultaneously solve the following system of equations, find the inverse of the coefficient matrix, and find the determinant of the coefficient matrix.

$$14X + 2Y - 6Z = 9$$

$$-4X + Y + 9Z = 3$$

$$6X - 4Y + 3Z = -4$$

The matrix to be entered will consist of the original coefficient matrix augmented by the identity matrix and augmented by the final column of constants. This is a 3x7 matrix.

$$\left[\begin{array}{ccc|ccc|c} 14 & 2 & -6 & 1 & 0 & 0 & 9 \\ -4 & 1 & 9 & 0 & 1 & 0 & 3 \\ 6 & -4 & 3 & 0 & 0 & 1 & -4 \end{array}\right]$$

Press	Function	See in Display
A	Initialization for a new matrix	"START REG. ?"
10 R/S	Start storing matrix in R10 and beyond	"DIM: R?↑C?"
3 ENTER↑ 7 R/S	Key in dimension as 3 rows and 7 columns	-TONE 9- "(1,1)=?"

Next enter the rows of the above matrix as directed by the display. Then simply XEQ "RRM". When the program ends (about 49 seconds) key CAT 1 R/S. Then press B to display the matrix:

$$\left[\begin{array}{ccc|ccc|c} 1 & 0 & 0 & 0.0631 & 0.0291 & 0.0388 & 0.5000 \\ 0 & 1 & 0 & -0.1068 & 0.1262 & -0.1650 & 2.0000 \\ 0 & 0 & 1 & 0.0162 & 0.1100 & 0.0356 & 0.3333 \end{array}\right]$$

The inverse of the original matrix is the 3x3 matrix in the middle. The ROM routine **DF** can be used to convert these decimals to fractions. For the mathematical purist who then wishes to see the exact inverse:

$$\begin{bmatrix} 13/206 & 3/103 & 4/103 \\ 11/103 & 13/103 & -17/103 \\ 5/309 & 34/309 & 11/309 \end{bmatrix}$$

The last column contains the solutions of the system of equations and would be interpreted as X=1/2, Y=2, Z=1/3. The determinant of the coefficient matrix can be recalled from R01. det. = 618.

Some final comments about RRM are in order. If you are only interested in the determinant of a matrix then a square matrix is all that RRM requires. In this case the matrix need not be augmented by any extra columns. RRM always leaves the determinant in R01 but this can be changed to any register by changing lines 06, 34, 50, and 52 in the RRM listing.

Systems of equations are solved as in problem 1, inverses are solved as in problem 2, and the combination of inverse and a system of equations is solved as in problem 3. RRM is just as useful for systems of equations which do not have unique solutions. If the determinant in R01 is 0 (or is so small as to be considered 0) then the system of equations may have no solutions or an infinite number of solutions. Since RRM returns the row reduced echelon form, the final matrix will always be row equivalent to the original. The final matrix may then be used to tell immediately where parameters should be inserted and any and all solutions may then be immediately determined. The coefficient matrix need not be square for RRM to operate on it.

Line By Line Analysis of RRM:

Lines 02-07 initialize the program by storing a 1 in R01 for the determinant and setting flag F10 for the **BX** routine.

Lines 08-12 make R03 & R04 point to the next pivot position.

Lines 13-20 determine when the program ends by checking if either a row or column boundary has been exceeded.

Lines 21-31 set up the block control word for the **BX** routine.

Lines 32-36 find the pivot number and check if all the remaining column entries are zero in which case the determinant must be zero and only the next column is incremented by branching to LBL 06.

Lines 37-43 make a 1 in the row containing the pivot number.

Lines 44-48 check if the pivot number is already in the pivot position. Lines 049-052 perform a row interchange to move the pivot to the true pivot position and adjust the sign of the determinant accordingly.

Lines 53-70 make 0's in the current pivot column in all rows except the pivot row.

BAR CODE ON PAGE 480

APPLICATION PROGRAM FOR: **M1**

```
01•LBL "RRM"
02 .
03 STO 03
04 STO 04
05 SIGN
06 STO 01
07 SF 10
08•LBL 05
09 ISG 03
10•LBL 06
11 ISG 04
12 ""
13 RCL 08
14 RCL 04
15 X>Y?
16 RTN
17 RCL 09
18 RCL 03
19 X>Y?
20 RTN
21 RCL 04
22 XROM "M5"
23 X<> Z
24 XROM "M5"
25  E3
26 /
27 +
28 RCL 08
29  E5
30 /
31 +
32 XROM "BX"
33 RCL IND [
34 ST* 01
35 X=0?
36 GTO 06
37 1/X
38 RCL [
39 INT
40 XROM "M4"
41 RDN
42 STO 02
43 XROM "M2"
44 RCL 02
45 ST- 02
46 RCL 03
47 X=Y?
48 GTO 07
49 XROM "M1"
50 RCL 01
51 CHS
52 STO 01
53•LBL 07
54 ISG 02
55 ""
56 RCL 09
57 RCL 02
58 X>Y?
59 GTO 05
60 RCL 03
61 X=Y?
62 GTO 07
63 RCL 02
64 RCL 04
65 XROM "M5"
66 RDN
67 RCL IND T
68 CHS
69 XROM "M3"
70 GTO 07
```

BAR CODE ON PAGE 480

APPLICATION PROGRAM FOR: **M1**

```
01♦LBL "MIO"
02♦LBL A
03 "START REG. ?"
04 AVIEW
05 STOP
06 STO 07
07 "DIM: R?↑C?"
08 AVIEW
09 STOP
10 STO 08
11 X<>Y
12 STO 09
13 SF 09
14 GTO 01
15♦LBL B
16 CF 09
17♦LBL 01
18 CF 29
19 RCL 07
20 STO 04
21 RCL 08
22 RCL 09
23 *
24 STO 03
25♦LBL 02
26 RCL 04
27 XROM "M4"
28 FIX 0
29 " ("
30 ARCL Y
31 "⊢,"
32 ARCL X
33 "⊢)="
34 FC? 09
35 GTO 03
36 "⊢?"
37 AVIEW
38 TONE 9
39 STOP
40 STO IND 04
41 GTO 04
42♦LBL 03
43 FIX IND 05
44 ARCL IND 04
45 AVIEW
46♦LBL 04
47 ISG 04
48 ""
49 DSE 03
50 GTO 02
51 BEEP
52 RTN
53♦LBL C
54 XROM "M5"
55 " R"
56 ARCL X
57 AVIEW
58 STO 04
59 E
60 STO 03
61 CF 09
62 GTO 02
```

Routine Listing For: **M1**

```
28♦LBL "M1"
29 XEQ 00
30 X<>Y
31 XEQ 00
32♦LBL "BE"
33 RCL IND Y
34 X<> IND Y
35 STO IND Z
36 RDN
37 ISG X
38 ""
39 ISG Y
40 GTO "BE"
41 RTN
42♦LBL 00
43 RCL 08
44 *
45 RCL 07
46 +
47 RCL X
48 RCL 08
49 ST- Z
50 SIGN
51 -
52 E3
53 /
54 +
55 RTN
```

LINE BY LINE ANALYSIS OF M1

M1 feeds into the block exchange routine **BE** after setting up the two block control words for the two rows by calling the local label LBL 00 twice. If R07=s=starting register of the matrix and R08=c=the number of columns in the matrix and i=the row number of the ith row, then with i in X, LBL 00 computes bbb.eee=the block control word for row i.

bbb = s + c*(i-1)   eee = s + c*i - 1

CONTRIBUTORS HISTORY FOR M1

The **M1** routine and documentation are by John Kennedy (918).

FURTHER ASSISTANCE ON M1

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

T E C H N I C A L   D E T A I L S

XROM: 20, 33 — **M1** — SIZE: depends on matrix size

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 4

ΣREG: not used

Data Registers:
- R00: not used
- R06: not used
- R07: s=start reg. matrix
- R08: c=# columns in matrix
- R09: not used
- R10: not used
- R11: not used
- R12: not used

Global Labels Called:

Direct	Secondary
none	none
falls into **BE** routine	

Local Labels In This Routine: 00

Execution Time: depends on matrix size. 1.07C + 0.56 seconds where C = # columns in matrix

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: **M2**

Bytes In RAM: 56

Registers To Copy: 61

Other Comments:

M2 - MATRIX, MULTIPLY ROW BY CONSTANT

M2 is the second of five matrix routines. M2 will multiply a row in a matrix by a constant. The normal input to M2 is simply the constant and the row number. M2 may also be considered along with the IR and DR routines which are part of a data base management system. If the records consist of numerical entries then M2 may be used to multiply a record by a constant. In this case the input to M2 is considered to be the constant and the record number. Choosing the constant as zero will clear a record.

In addition, each of the five matrix routines requires two stored values, one of which is the starting register of the matrix and the other is the number of columns in the matrix. Matrices are assumed to be stored with each row occupying a block of consecutive registers. Thus the number of columns is the block size and the entire matrix is stored row by row as one string of consecutive registers. R07 holds the starting register of the matrix and R08 holds the number of columns.

Example 1: Use M2 to double the last row of the matrix:

21	35	55	74	83
11	93	56	36	29
65	78	32	27	75
53	94	46	62	97
54	39	61	67	82
23	45	77	15	25

In this example we will assume the matrix is stored in registers R15-R44 so that R07=15 and R08=5. See the first example of the routine M1 for an indication of exactly how the matrix elements are to be stored.

To multiply the 6th row by the constant 2, key 2 ENTER↑ 6 and XEQ "M2".

The new matrix now contains the following elements.

21	35	55	74	83
11	93	56	36	29
65	78	32	27	75
53	94	46	62	97
54	39	61	67	82
46	90	154	30	50

Inspect registers R40-R44 and you should see the following numbers in those registers:

R40: 46 R41: 90 R42: 154 R43: 30 R44: 50

COMPLETE INSTRUCTIONS FOR M2

1) The matrix is assumed to be stored with each row occupying a consecutive block of registers. The entire matrix is assumed to be stored row by row as one string of consecutive data registers.

2) The number of the starting register for the matrix is to be stored in R07. The number of columns in the matrix is to be stored in R08. Both the row and column numbers start counting from 1.

3) To multiply row i by the constant k, key k ENTER↑ i and XEQ "M2". The following is the stack input/output for M2.

Input:		Output:	
T:	T	T:	k
Z:	Z	Z:	k
Y:	k=constant	Y:	*
X:	i=row number	X:	k
L:	L	L:	*

MORE EXAMPLES OF M2

Example 2: Use M2 to multiply row 3 in the following matrix by the constant -5.

The matrix is in the following form:

32	54	67	89	55
21	63	81	35	45
6	15	19	14	16
13	72	49	57	72
42	53	68	19	82
44	90	61	33	15
36	25	41	56	27

We may store this matrix in any block of consecutive registers as long as R07 contains the starting register number and R08 contains the number of columns. Store 20 in R07 and store 5 in R08. Then begin storing the matrix elements in R20. The elements should be stored row by row in consecutive registers. The last element will be in R54.

To multiply row 3 by -5, key 5 CHS ENTER↑ 3 and XEQ "M2". Then recall the data from the following registers to insure that the proper operation has been carried out.

R30: -30 R31: -75 R32: -95 R33: -70 R34: -80

Example 3: In this example we have a file which contains a list of materials prices for various models of houses which are part of a special construction project. Use M2 to increase the cost of the materials of House #4 by 10%.

In this example we can assume the prices are arranged as rows of a matrix.

	Concrete	Lumber	Brick	Shingles
House 1	$10	$14	$20	$18
House 2	$34	$36	$30	$40
House 3	$25	$30	$50	$32
House 4	$18	$42	$28	$24

House 5	$33	$48	$20	$21
House 6	$19	$35	$29	$39
House 7	$29	$34	$34	$42

As in other file related routines (such as IR and DR) we store the records of the file (in this case the prices of materials for the various houses) in consecutive registers. The starting register for the entire file is to be stored in R07. For this example we will assume the file starts in R25. Store 25 in R07. The number of registers per record (in this case there are 4 prices per house) should be stored in R08. Store 4 in R08. The total number of records (in this case there are 7 houses) should be stored in R09. Store 7 in R09 and store the remaining prices starting in register 25 as in the above matrix form. The last entry, 42, should be stored in R52. M2 does not use the number in R09.

This example simply requires multiplying the fourth record (row) by the constant 1.10. Key 1.1 ENTER↑ 4 and XEQ "M2". The following registers should be interpreted as containing the following prices.

R37: $19.80 R38: $46.20 R39: $30.80 R40: $26.40

APPLICATION PROGRAM 1 FOR M2

See the RRM program in the M1 routine documentation.

Routine Listing For: **M2**

```
01♦LBL "M2"
02 XEQ 00
03 X<>Y
04♦LBL 01
05 ST* IND Y
06 ISG Y
07 GTO 01
08 RTN

42♦LBL 00
43 RCL 08
44 *
45 RCL 07
46 +
47 RCL X
48 RCL 08
49 ST- Z
50 SIGN
51 -
52  E3
53 /
54 +
55 RTN
```

LINE BY LINE ANALYSIS OF M2

M2 calls local label LBL 00 which sets up the block control word for row i which is of the form bbb.eee where:

bbb = s + c*(i-1)   eee = s + c*i - 1

R07=s=starting register of the matrix
R08=c=number of columns in the matrix

The constant k is then placed in X and M2 runs through a short loop to multiply each element of row i by the constant k.

CONTRIBUTORS HISTORY FOR M2

The M2 routine and documentation are by John Kennedy (918).

FURTHER ASSISTANCE ON M2

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

TECHNICAL DETAILS

XROM: 20, 31 — **M2** — SIZE: depends on matrix size

Stack Usage:	Flag Usage:
0 T: used	04: not used
1 Z: used	05: not used
2 Y: used	06: not used
3 X: used	07: not used
4 L: used	08: not used
Alpha Register Usage:	09: not used
5 M: not used	10: not used
6 N: not used	
7 O: not used	
8 P: not used	25: not used
Other Status Registers:	**Display Mode:**
9 Q: not used	not used
10 ⊢: not used	
11 a: not used	**Angular Mode:**
12 b: not used	not used
13 c: not used	
14 d: not used	**Unused Subroutine Levels:**
15 e: not used	4
ΣREG: not used	**Global Labels Called:**
Data Registers:	Direct: none — Secondary: none
R00: not used	
R06: not used	
R07: s=start reg. matrix	
R08: c=# columns	
R09: not used	
R10: not used	
R11: not used	**Local Labels In This Routine:**
R12: not used	00, 01

Execution Time: depends on matrix size.
0.16C + 0.86 seconds where C = # columns in matrix

Peripherals Required: none

Interruptible? yes	Other Comments:
Execute Anytime? no	
Program File: M2	
Bytes In RAM: 38	
Registers To Copy: 61	

M3 - MATRIX, ADD MULTIPLE OF ANOTHER ROW

M3 is the third of five matrix routines. **M3** will add a constant multiple of one row in a matrix to another row. The row that is multiplied by the constant does not change. **M3** may also be considered part of the data base system routines **IR** and **DR**. When records consist of numerical entries (such as rows of prices) **M3** may be used to add a multiple of one record to another.

In addition, each of the five matrix routines requires two stored values, one of which is the starting register of the matrix and the other is the number of columns in the matrix. Matrices are assumed to be stored with each row occupying a block of consecutive registers. Thus the number of columns is the block size and the entire matrix is stored row by row as one string of consecutive registers. R07 holds the starting register of the matrix and R08 holds the number of columns.

Example 1: Use **M3** to add -2 times row 3 to row 4 in the following matrix.

21	35	55	74	83
11	93	56	36	29
65	78	32	27	75
53	94	46	62	97
54	39	61	67	82
23	45	77	15	25

In this example we will assume the matrix is stored in registers R15-R44 so that R07=15 and R08=5. See the first example of the routine **M1** for an indication of exactly how the matrix elements are to be stored.

To add -2 times row 3 to row 4 key 4 ENTER↑ 3 ENTER↑ 2 CHS and XEQ "**M3**". The stack should contain the following when **M3** is called.

T: *
Z: j = row number of the row to be changed
Y: i = row number of the row to be multiplied
X: k = constant multiplying row i

The new matrix now contains the following elements.

21	35	55	74	83
11	93	56	36	29
65	78	32	27	75
-77	-62	-18	8	-53
54	39	61	67	82
23	45	77	15	25

Inspect registers R30-R34 and you should see the following numbers in those registers:

R30:-77 R31:-62 R32:-18 R33: 8 R34:-53

COMPLETE INSTRUCTIONS FOR M3

1) The matrix is assumed to be stored with each row occupying a consecutive block of registers. The entire matrix is assumed to be stored row by row as one string of consecutive data registers.

2) The number of the starting register for the matrix is to be stored in R07. The number of columns in the matrix is to be stored in R08. Both the row and column numbers start counting from 1.

3) To add k times row i to row j, key j ENTER↑ i ENTER↑ k and XEQ "**M3**". The following is the stack input/output for **M3**.

Input:
T: T
Z: j = row number of the row to be changed
Y: i = row number of the row to be multiplied
X: k = constant multiplying row i

L: L

Output:
T: *
Z: *
Y: *
X: *

L: k

MORE EXAMPLES OF M3

Example 2: Use **M3** to add -2 times row 3 to row 1 in the following matrix. Then perform the operation of adding 2 times row 3 to row 1 to undo the first operation.

12	31	-7	-6	64
51	23	73	91	14
42	26	-34	11	17
31	16	49	47	22
64	25	33	60	19

Store the entries of this matrix in registers R20-R44. Then store 20 in R07=starting register of matrix and store 5 in R08=number of columns.

To perform the operation key 1 ENTER↑ 3 ENTER↑ 2 CHS and then XEQ "**M3**". The row 1 entries should now appear in R20-R24 as:

R20:-72 R21:-21 R22:61 R23:-28 R24:30

Next, we will undo the operation we have just completed by performing +2 times row 3 and add this to row 1 to change row 1 back to its original content.

Key 1 ENTER↑ 3 ENTER↑ 2 and XEQ "**M3**".

Row 1 should now appear in R20-R24 as:

R20:12 R21:31 R22:-7 R23:-6 R24:64

APPLICATION PROGRAM 1 FOR M3

See the support program RRM in the M1 routine documentation.

Routine Listing For: M3	
09♦LBL "M3" 10 STO [11 RDN 12 XEQ 00 13 X<>Y 14 XEQ 00 15 RCL [16 SIGN 17♦LBL 02 18 RDN 19 RCL IND Y 20 LASTX 21 * 22 ST+ IND Y 23 ISG Y 24 "" 25 ISG Z 26 GTO 02 27 RTN	42♦LBL 00 43 RCL 08 44 * 45 RCL 07 46 + 47 RCL X 48 RCL 08 49 ST- Z 50 SIGN 51 - 52 E3 53 / 54 + 55 RTN

LINE BY LINE ANALYSIS OF M3

Lines 09-16 initialize the M3 routine for the loop that starts with label 02. The M register is used for temporary storage. Next M3 calls local label LBL 00 to set up the block control words for the two rows involved in the matrix operation. See the documentation for M1 for details on the LBL 00 subroutine. The SIGN function at line 16 stores the constant k in LAST X.

Lines 17-27 are the main loop in the program. The contents of the stack at LBL 02, line 17 are:

T: scratch
Z: block control word for row i
Y: block control word for row j
X: scratch
L: constant k

CONTRIBUTORS HISTORY FOR M3

The M3 routine and documentation are by John Kennedy (918).

FURTHER ASSISTANCE ON M3

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

TECHNICAL DETAILS

XROM: 20, 32 — M3 — SIZE: depends on matrix size

Stack Usage:
0 T: used
1 Z: used
2 Y: used
3 X: used
4 L: used

Alpha Register Usage:
5 M: temp. holds k
6 N: not used
7 O: not used
8 P: not used

Other Status Registers:
9 Q: not used
10 ⊢: not used
11 a: not used
12 b: not used
13 c: not used
14 d: not used
15 e: not used

ΣREG: not used

Data Registers:
R00: not used

R06: not used
R07: s=start reg. matrix
R08: c=# columns matrix
R09: not used
R10: not used
R11: not used
R12: not used

Flag Usage:
04: not used
05: not used
06: not used
07: not used
08: not used
09: not used
10: not used

25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine: 00, 02

Execution Time: depends on matrix size
0.33C + 1.33 seconds where C = # columns in matrix

Peripherals Required: none

Interruptible? yes
Execute Anytime? no
Program File: M2
Bytes In RAM: 55
Registers To Copy: 61

Other Comments:

M4 - MATRIX, REGISTER ADDRESS TO (i,j)

M4 is the fourth of five matrix routines. M4 will determine the (i,j) element in a matrix (row i and column j), given the number of the data register which contains that element. M4 is the inverse of the routine M5. The normal input to M4 is simply the register number. M4 may also be considered part of the file management routines IR and DR. M4 can be used to determine a particular field element in a record. Input in this case is also the number of the register which contains the desired item.

In addition, each of the five matrix routines requires two stored values, one of which is the starting register of the matrix and the other is the number of columns in the matrix. Matrices are assumed to be stored with each row occupying a consecutive block of registers. Thus the number of columns is the block size and the entire matrix is stored row by row as one string of consecutive registers. R07 holds the starting register of the matrix and R08 holds the number of columns.

Example 1: The following 6x5 matrix is assumed to be stored in registers R15-R44. Use M4 to determine the row and column numbers of the element stored in register number 38. This matrix was used in Example 1 in the M1 routine.

21	35	55	74	83
11	93	56	36	29
65	78	32	27	75
53	94	46	62	97
54	39	61	67	82
23	45	77	15	25

For this example we will explicitly show the correspondence between the data registers and the matrix elements. The element in the upper left-hand corner is assumed to be in row 1 and column 1. Store the matrix entries in the following registers.

R15: 21	R23: 36	R31: 94	R39: 82
R16: 35	R24: 29	R32: 46	R40: 23
R17: 55	R25: 65	R33: 62	R41: 45
R18: 74	R26: 78	R34: 97	R42: 77
R19: 83	R27: 32	R35: 54	R43: 15
R20: 11	R28: 27	R36: 39	R44: 25
R21: 93	R29: 75	R37: 61	
R22: 56	R30: 53	R38: 67	

Since this matrix starts in R15 and the number of columns in the matrix is 5 we must first store the following in R07 and R08.

R07: 15=starting register   R08: 5=# of columns

Any number of matrix operations may be performed on the above matrix without changing the numbers in R07 and R08. These operations include M1, M2, M3, M4, or M5.

Since most matrix algorithms are written in mathematical notation which depends on the subscripts i and j it is convenient to have a routine to calculate the (i,j) subscripts, given the data register number which holds that element. M4 does not perform any useful operations on the matrix but M4 is a useful utility routine for any program which requires storing, retreiving, or finding particular elements of a matrix.

Now to determine the row and column of the element stored in register 38, key in 38 and XEQ "M4". The above matrix does not change since M4 does not perform any operations on the matrix elements. After executing M4 the Y register will contain the row number of the element and the X register will contain the column number of the element. In this example see 4 returned in X and see 5 returned in Y. The element in register R38 is the (5,4) element.

COMPLETE INSTRUCTIONS FOR M4

1) The matrix is assumed to be stored with each row occupying a consecutive block of registers. The entire matrix is assumed to be stored row by row as one string of consecutive data registers.

2) The number of the starting register for the matrix is to be stored in R07. The number of columns in the matrix is to be stored in R08. Both the row and column numbers start counting from 1.

3) To determine the (i,j) address of the matrix element stored in register r, key r in X and XEQ "M4". The row and column numbers will be returned in Y and X.

Input:		Output:	
	T: T		T: Z
	Z: Z		Z: Y
	Y: Y		Y: row number
	X: register number		X: column number
	L: L		L: R08

MORE EXAMPLES OF M4

Example 2: Use the matrix in Example 1 and find the matrix elements in registers R25 and R34.

Assuming the matrix from Example 1 is still in the machine key 25 XEQ "M4". The stack should return:

Y: 3
X: 1   R25 holds the (3,1) element.

Next, key in 34 XEQ "M4". The stack returns:

Y: 4
X: 5   R34 holds the (4,5) element.

Example 3: The following list of registers shows an example file consisting of a simplified telephone directory. Use M4 to determine what is stored in registers R25, R29, and R39.

This is the example file in Example 1 of IR and DR. This example file consists of a list of names and phone numbers. Only six records are in the file to begin with. Each record consists of 6 consecutive registers with the following format:

1st register holds first name
2nd and 3rd registers hold the last name
4th register holds the telephone number
5th register holds the city name
6th register holds the state name

The records of the original file are assumed to be the following:

Record #1:	Mary Adams 354-1662 Gary, IN
Record #2	Jane Hamilton 363-5648 Boston, MA
Record #3	Robert Jefferson 261-2347 Fresno, CA
Record #4	Mike Johnson 745-3254 Denver, CO
Record #5	James Masterson 565-2314 Toledo, OH
Record #6	Joe Robinson 756-4438 Peoria, IL

This sample file is stored in data registers R10-R45 where each record consists of 6 consecutive data registers.

R10: Mary	R28: Mike
R11: Adams	R29: Johnso
R12:	R30: n
R13: 354.1662	R31: 745.3254
R14: Gary	R32: Denver
R15: IN	R33: CO
R16: Jane	R34: James
R17: Hamilt	R35: Master
R18: on	R36: son
R19: 363.5648	R37: 565.2314
R20: Boston	R38: Toledo
R21: MA	R39: OH
R22: Robert	R40: Joe
R23: Jeffer	R41: Robins
R24: son	R42: on
R25: 261.2347	R43: 756.4438
R26: Fresno	R44: Peoria
R27: CA	R45: IL

When considered part of the data base management routines, M4 can expect to find the following information in registers R07, R08, and R09, even though M4 does not use the number in R09.

R07: starting register of entire file
R08: number of registers per record
R09: total number of records in the file

For the above sample file these numbers are:

R07: 10 = starting register
R08: 6 = number of registers per record
R09: 6 = total number of records

Now to determine what is in register R25, key in 25 and XEQ "M4". The X and Y registers return with:

Y: 3 = 3rd record
X: 4 = 4th item in the record

The numbers in the X and Y registers should be interpreted as pointing to the 4th item in the 3rd record. Knowing that the 4th item in all records in this example holds the telephone number, we would know that R25 holds the telephone number of person #3.

Next key in 29 and XEQ "M4". The X and Y registers return with:

Y: 4 = 4th record
X: 2 = 2nd item in the record

Knowing that the 2nd item in each record in this example is the start of the last name we would interpret these results as saying that in R29 is the first six characters of the last name of the 4th person in the file.

Finally key in 39 and XEQ "M4". The X and Y registers return with:

Y: 5 = 5th record
X: 6 = 6th item in the record

Since the 6th item of each record is the two-character name of the state of each person we would interpret these numbers as saying that in R39 we should find the name of the state of the 5th person in the file.

APPLICATION PROGRAM 1 FOR M4

See the RRM and M10 programs in the M1 routine documentation.

Routine Listing For:	M4
56♦LBL "M4" 57 RCL 07 58 - 59 RCL 08 60 XROM "QR"	61 ISG Y 62 "" 63 ISG X 64 "" 65 RTN

LINE BY LINE ANALYSIS OF M4

The M4 routine determines the row number i and the column number j from the register number r by the following formulas where s=starting register of the matrix and c=the number of columns in the matrix:

$$i = INT((r-s)/c) + 1$$

$$j = (r-s) \text{ MOD } c + 1$$

Line 60 calls on the quotient-remainder routine QR to calculate i-1 and j-1 in one step. These results are left in the X and Y registers and then incremented in lines 61 and 63. Lines 62 and 64 are NOP's.

CONTRIBUTORS HISTORY FOR M4

The M4 routine and documentation are by John Kennedy (918).

FURTHER ASSISTANCE ON M4

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 evenings

NOTES

TECHNICAL DETAILS

XROM: 20, 35	M4	SIZE: depends on matrix size

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: scratch in QR
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 4

ΣREG: not used

Data Registers:
- R00: not used
- R06: not used
- R07: s=start reg. matrix
- R08: c=# columns matrix
- R09: not used
- R10: not used
- R11: not used
- R12: not used

Global Labels Called:

Direct	Secondary
QR	none

Local Labels In This Routine: none

Execution Time: 2 seconds

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: M2

Bytes In RAM: 21

Registers To Copy: 61

Other Comments:

APPENDIX G CONTINUED FROM PAGE 251.

TERMINATION - The process that the HP-41 (and most calculators) performs when digit (or ALPHA) entry is complete. ENTER, a function key, etc. terminates digit entry. A terminated display (input) is cleared when a digit key is pressed. Termination may be done by : OFF/ON, ALPHA/ALPHA, PRGM/PRGM, etc. This is necessary for entry of two consecutive numbers in a program.

V

VOID - See Memory Void.

X

X REGISTER - The fourth register (003 absolute) of HP-41 memory. The X register is the bottom register of the four register (RPN) stack--XYZT. The X register is the normal Display register. Also see Status Registers.

XROM - External ROM used in HP-41 ports. An XROM identification system used by the HP-41 operating system provides for an A, B two byte "word" of the form: A is 0 thru 31 for 31 unique ROM's. B is function number in the ROM. XROM numbers may be real, as obtained from the ROM data, or psuedo as many synthetic key assignments show. See Psuedo XROM.

Y

Y REGISTER - The third register (002 absolute) of HP-41 memory. The Y register is the third register from the top of the four register (RPN) Stack--XYZT. Also see Status Registers.

Z

Z REGISTER - The second register (001 absolute) of HP-41 memory. The Z register is the second register from the top of the four register (RPN) Stack--XYZT. Also see Status Registers.

MISC.

41 "LANGUAGE" - Normal X $\rightleftarrows$ Y, STO 10, etc. instructions used in programming the HP-41. This is in contrast to Assembly Language ("microcode") or Machine Language (binary).

a REGISTER - The twelfth register (011 absolute) of HP-41 memory. This register is used for subroutine return address storage for the 3rd (one byte, half), 4th, 5th, and 6th levels of subroutines. If a program doesn't require more than two levels of subroutines, the a register may be used for general purpose storage.

b REGISTER - The thirteenth register (012 absolute) of HP-41 memory. This register is used by the 41 to store the 1st, 2nd, and half of 3rd return address and the address pointer. This register provides a powerful tool for increasing program execution speed (pre-compiling) and arbitrary entry into RAM or ROM.

c REGISTER - The fourteenth register (013 absolute) of HP-41 memory. This register contains the location of ΣREG, R00 (curtain), and .END.. Eight BITS are used for scratch and twelve BITS are used for a "Cold Start" value, that, if not 169 HEX, causes a Master Clear to be performed. 0, STO c in a program clears all program and data memory.

d REGISTER - The fifteenth register (014 absolute) of HP-41 memory. This register is used for all user flags--F00 through F55. All flags may set or clear as desired by a single STO d instruction if the "right" nnn is in the X register.

e REGISTER - The sixteenth register (015 absolute) of HP-41 memory. This register is used for shifted key assignment map, scratch, and program line number counter.

ͱ REGISTER - The eleventh register (010 absolute) of HP-41 memory. This register is used for the unshifted key assignment map (36 BITS) and scratch.

NOTES

END

M5 - MATRIX, (i,j) TO REGISTER ADDRESS

M5 is the fifth of five matrix routines. M5 will determine the register number of the (i,j) element in a matrix. (row i and column j). M5 is the inverse of the routine M1. The normal input to M5 is simply the row number i and the column number j. M5 may also be considered part of the file management routines IR and DR. M5 can be used to locate a particular field element in a record. Input in this case is the record number and the number of the desired item within the record.

In addition, each of the five matrix routines requires two stored values, one of which is the starting register of the matrix and the other is the number of columns in the matrix. Matrices are assumed to be stored with each row occupying a consecutive block of registers. Thus the number of columns is the block size and the entire matrix is stored row by row as one string of consecutive registers. R07 holds the starting register of the matrix and R08 holds the number of columns.

Example 1: Use M5 to determine the register number of the (2,3) element in the following 6x5 matrix which is assumed to be stored in registers R15-R44.

21	35	55	74	83
11	93	56	36	29
65	78	32	27	75
53	94	46	62	97
54	39	61	67	82
23	45	77	15	25

This matrix is from Example 1 in the M1 routine. For this example we will explicitly show the correspondence between the data registers and the matrix elements. The element in the upper left-hand corner is assumed to be in row 1 and column 1. Store the matrix entries in the following registers.

R15: 21	R23: 36	R31: 94	R39: 82
R16: 35	R24: 29	R32: 46	R40: 23
R17: 55	R25: 65	R33: 62	R41: 45
R18: 74	R26: 78	R34: 97	R42: 77
R19: 83	R27: 32	R35: 54	R43: 15
R20: 11	R28: 27	R36: 39	R44: 25
R21: 93	R29: 75	R37: 61	
R22: 56	R30: 53	R38: 67	

Since this matrix starts in R15 and the number of columns in the matrix is 5 we must first store the following in R07 and R08.

R07: 15=starting register   R08: 5=# of columns

Any number of matrix operations may be performed on the above matrix without changing the numbers in R07 and R08. These operations include M1, M2, M3, M4, or M5.

Since most matrix algorithms are written in mathematical notation which depends on the subscripts i and j it is convenient to have a routine to calculate the register address of the (i,j) element. M5 does not perform any useful operations on the matrix but M5 is a useful utility routine for any program which requires storing, retreiving, or finding particular elements of a matrix.

Now to determine the register number for the (2,3) element key 2 ENTER↑ 3 and XEQ "M5". The above matrix does not change since M5 does not perform any operations on the matrix elements. However, the X register should now contain the number 22 which is the register holding the (2,3) element 56. To check this key RCL IND X and see the number 56 returned in X.

COMPLETE INSTRUCTIONS FOR M5

1) The matrix is assumed to be stored with each row occupying a consecutive block of registers. The entire matrix is assumed to be stored row by row as one string of consecutive data registers.

2) The number of the starting register for the matrix is to be stored in R07. The number of columns in the matrix is to be stored in R08. Both the row and column numbers start counting from 1.

3) To determine the register number of the (i,j) element in the matrix, key i ENTER↑ j and XEQ "M5". The register number will be returned in X. Note that the order of the input is important. The row number should be in Y and the column number should be in X. The stack input/output for M5 is as follows.

Input:		Output:	
	T: T		T: T
	Z: Z		Z: T
	Y: row #		Y: Z
	X: column #		X: register #
	L: L		L: R07

MORE EXAMPLES OF M5

Example 2: Use the matrix in Example 1 and find the register numbers of the elements (4,5) and (6,1).

Assuming the matrix from Example 1 is still in the machine simply key 4 ENTER↑ 5 and XEQ "M5". The register number returned in X is 34. Key RCL 34 and see the element 97 returned.

Next key 6 ENTER↑ 1 and XEQ "M5". See 40 returned. RCL 40 and see the number 23.

Example 3: Use the example file of the name and address list from Example 3 of the M4 routine for this example. Use M5 to recall the following information from the file.

Find the telephone numbers of Jane Hamilton and James Masterson.

Find the first name of the 4th person in the file.

Find the cities that Jane Hamilton and James Masterson live in.

When M5 is used as one of the data base management routines its input is considered to be a record number and an item number within the record.

Since Jane Hamilton is the 2nd person and since for this example file the telephone number is always the 4th item within a record, to find her telephone number key 2 ENTER↑ 4 and XEQ "M5". See the number 19 returned in X indicating that R19 holds the desired information. RCL IND X will produce the actual phone number = 363.5648

James Masterson is the 5th person in the file and his telephone number is found in a manner similar to that used for Jane Hamilton. Key 5 ENTER↑ 4 and XEQ "M5". See 37 returned indicating James Masterson's telephone number can be found in register R37. RCL IND X and see 565.2314

To use M5 to find the first name of the fourth person in the file simply key 4 ENTER↑ 1 and XEQ "M5". See 28 returned. RCL IND X and see the name Mike returned. Mike Johnson is the 4th person in the file.

Now to find the city that Robert Jefferson lives in note that cities are always the 5th item in each file and that Robert is the 3rd person in the file. Simply key 3 ENTER↑ 5 and XEQ "M5". The number 26 is returned and recalling the information in R26 reveals that Robert lives in Fresno.

Joe Robinson is the 6th person in the file and his city can be found by keying in 6 ENTER↑ 5 and XEQ "M5". His city is in R44. Recalling R44 yields Peoria as his city.

APPLICATION PROGRAM 1 FOR M5

See the RRM and MIO programs in the M1 routine documentation.

Routine Listing For: M5

```
66♦LBL "M5"        72 X<> 08
67 X<> 08          73 1
68 ST- 08          74 -
69 *               75 RCL 07
70 ST+ 08          76 +
71 X<> L           77 RTN
```

LINE BY LINE ANALYSIS OF M5

The M5 routine simply calculates the register number r from the following data.

i = row number (user input in Y)
j = column number (user input in X)
s = starting register of matrix (user supplied R07)
c = number of columns in matrix (user supplied R08)

r = s + c*(i-1) + (j-1)

CONTRIBUTORS HISTORY FOR M5

The M5 routine and documentation are by John Kennedy (918).

FURTHER ASSISTANCE ON M5

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

TECHNICAL DETAILS

XROM: 20, 36 — **M5** — SIZE: depends on matrix size

Stack Usage:
0 T: preserved in T,Z
1 Z: preserved in Y
2 Y: used
3 X: used
4 L: used

Alpha Register Usage:
5 M: not used
6 N: not used
7 O: not used
8 P: not used

Other Status Registers:
9 Q: not used
10 ⊢: not used
11 a: not used
12 b: not used
13 c: not used
14 d: not used
15 e: not used

ΣREG: not used

Data Registers:
R00: not used

R06: not used
R07: s=start reg. matrix
R08: c=# columns matrix
R09: not used
R10: not used
R11: not used
R12: not used

Flag Usage:
04: not used
05: not used
06: not used
07: not used
08: not used
09: not used
10: not used

25: not used

Display Mode:
not used

Angular Mode:
not used

Unused Subroutine Levels:
5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine:
none

Execution Time: 1 second

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: M2

Bytes In RAM: 24

Registers To Copy: 61

Other Comments:

MA - MEMORY TO ALPHA

The MA routine is used to store into ALPHA the contents of four (or less) data registers. The inverse routine, AM , is used to ALPHA STORE the contents of the ALPHA register into these data registers. Both routines require the standard bbb.eeeii block control word as the only input. Alpha is cleared prior to storing the data.

Example 1: Registers 13 thru 16 contain the data:

R13	ABCDEF
R14	GHIJKL
R15	MNOPQR
R16	STUVWX

Fill alpha with this data. FIX 3.

DO:	SEE:	RESULT:
13.016	13.016	Input for MA
XEQ MA	17.016	ISG went 13 to 17
ALPHA	A thru X	Data recalled.

Example 2: Store A thru L into ALPHA from R13 & R14.

DO:	SEE:	RESULT:
13.014	13.014	Input for MA
XEQ MA	15.014	Routine finished
ALPHA	A thru L	Data recalled

COMPLETE INSTRUCTIONS FOR MA

A bbb.eeeii "ISG" control number is required for MA to work properly. The ii part of this number may be used to assemble uniformly spaced register contents into the alpha register. See Example 3. Effective use of MA depends on the application. Typical use would be 1.004, XEQ MA for seven bytes of memory usage. The same result is obtained with CLA ARCL 01, ARCL 02, ARCL 03, ARCL 04 for nine bytes of memory usage. Example 4 illustrates an inefficient use of MA.

MORE EXAMPLES OF MA

Example 3: ERIC wants to send a list of the PPC ROM routines he had previously recorded on magnetic cards. Registers 1 thru 122 contain the two letter ROM Labels. Using BV and the 82143A printer ERIC listed the registers. The list, however, is too long. Why not use MA to produce a compact list? Here is how ERIC used MA .

First he wrote a routine to add a space after the two letters. See below. This made each register three characters which "spaces" nicely with the 24 character ALPHA register. The routine to the right of "ADD BLK" prints the list as shown below. MA works in this application because the block control number allows eight three character registers to be recalled to ALPHA just as easily as four six character registers.

Alternate lists, or displayed labels, may be obtained by setting flag 12 or by changing line 02 to 1.004 and line 06 to .004. The routine changes eliminate the display scroll.

```
01 LBL "ADD BLK"
02 1.122
03 LBL 01
04 CLA
05 ARCL IND X
06 "⊦ "
07 ASTO IND X
08 ISG X
09 GTO 01
10 RTN
```

```
01 122
02 1.008
03 LBL 02
04 XROM MA
05 XROM VA
06 .008
07 +
08 X<=Y
09 GTO 02
10 RTN
```

The 122 ROM Routine labels are not a multiple of 4, so Eric set his HP-41CV to SIZE 123. When the machine stopped with NONEXISTENT in the display he pressed ALPHA and PRINT to complete the last line. The various list combinations that Eric printed using the routine changes and setting flag 12 are reproduced below. Observe that the routine stops in ROM in some uses. Changing Line 05 to PRA increases execution speed because the display scroll is eliminated.

```
+K  -B  1K  2D
A?  AD  AL  AM
Ab  BA  BC  BD
BE  BI  BL  BM
BR  BV  BX  BΣ
C?  CA  CB  CD
CJ  CK  CM  CP
CU  CV  CX  DC
DF  DP  DR  DS
DT  E?  EP  EX
F?  FD  FI  FL
FR  GE  GN  HA
HD  HN  HP  HS
IF  IG  IP  IR
JC  L-  LB  LF
LG  LR  M1  M2
M3  M4  M5  MA
MK  ML  MP  MS
MT  NC  NH  NP
NR  NS  OM  PA
PD  PK  PM  PR
PO  PS  QR  RD
RF  RK  RN  RT
RX  Rb  S1  S2
S3  S?  SD  SE
SK  SM  SR  SU
SV  SX  Sb  T1
TB  TN  UD  UR
VA  VF  VK  VM
VS  XD  XE  XL
Σ?  ΣC
```

```
+K -B 1K 2D
A? AD AL AM
Ab BA BC BD
BE BI BL BM
BR BV BX BΣ
C? CA CB CD
CJ CK CM CP
CU CV CX DC
DF DP DR DS
DT E? EP EX
F? FD FI FL
FR GE GN HA
HD HN HP HS
IF IG IP IR
JC L- LB LF
LG LR M1 M2
M3 M4 M5 MA
MK ML MP MS
MT NC NH NP
NR NS OM PA
PD PK PM PR
PO PS QR RD
RF RK RN RT
RX Rb S1 S2
S3 S? SD SE
SK SM SR SU
SV SX Sb T1
TB TN UD UR
VA VF VK VM
VS XD XE XL
Σ? ΣC
```

```
+K -B 1K 2D A? AD AL AM
Ab BA BC BD BE BI BL BM
BR BV BX BΣ C? CA CB CD
CJ CK CM CP CU CV CX DC
DF DP DR DS DT E? EP EX
F? FD FI FL FR GE GN HA
HD HN HP HS IF IG IP IR
JC L- LB LF LG LR M1 M2
M3 M4 M5 MA MK ML MP MS
MT NC NH NP NR NS OM PA
PD PK PM PR PO PS QR RD
RF RK RN RT RX Rb S1 S2
S3 S? SD SE SK SM SR SU
SV SX Sb T1 TB TN UD UR
VA VF VK VM VS XD XE XL
Σ? ΣC
```

Example 4: Registers 10 and 20 are to be moved memory to ALPHA using MA. The program lines required to do this are compared with a non-ROM approach. This comparison shows that MA is not always Byte efficient if incorrectly used.

ROM	NON-ROM
10.02010 XEQ MA	CLA ARCL 10 ARCL 20
(10 Bytes)	(5 Bytes)

Routine Listing For: MA

```
44♦LBL "MA"
45 CLA
46♦LBL 02
47 ARCL IND X
48 ISG X
49 GTO 02
50 RTN
51 END
```

LINE BY LINE ANALYSIS OF MA

Line 45 clears ALPHA to accomplish the basic objective of using MA and (AM) to store the contents of the ALPHA register for future use. LINE 46 labels the recall loop. Line 47 ALPHA recalls the register defined by the integer of the block control number. Line 48 increments and tests the x register. If eee value is not reached line 49 causes the loop to repeat. The routine finishes with the pointer in the ROM.

CONTRIBUTORS HISTORY FOR MA

MA, like AM was conceived by Keith Jaret (4360) and Richard Nelson (1) during an early morning SDS loading session.

FINAL REMARKS FOR MA

See this section under AM .

FURTHER ASSISTANCE ON MA

Call Keith Jaret (4360) at (213)374-2583 eve.
Call Richard Nelson (1) at (714)754-6226 (P.M.)

NOTES

TECHNICAL DETAILS

XROM: 20,54 **MA** SIZE: ≥ 005

Stack Usage:
- 0 T: NOT USED
- 1 Z: NOT USED
- 2 Y: NOT USED
- 3 X: INPUT, ISG COUNTER
- 4 L: NOT USED

Alpha Register Usage:
- 5 M: USED
- 6 N: USED
- 7 O: USED
- 8 P: USED

Other Status Registers: NONE USED
- 9 Q:
- 10 ├:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

ΣREG: NOT USED

Data Registers:
R00: NOT USED
R06: R07: R08: R09: R10: R11: R12: 1 to 4 Registers selected by user.

Flag Usage: NONE
04: 05: 06: 07: 08: 09: 10: 25:

Display Mode: N/A

Angular Mode: N/A

Unused Subroutine Levels: 5

Global Labels Called:
Direct / Secondary: NONE

Local Labels In This Routine:
LBL 02

Execution Time: 1 second.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: NS
Bytes In RAM: 18
Registers To Copy: 16

Other Comments:

MK - MAKE MULTIPLE KEY ASSIGNMENTS

MK extends the capabilities of the ASN function to arbitrary one or two-byte codes. This gives handy access to most useful synthetic instructions, ones like STO M, X<>d, byte jumpers, byte maskers, Q-loaders, and their kin. MK is one of several PPC ROM programs that lessen the user's dependence on status cards, enhancing the ability of users without card readers to do synthetic programming (and greatly extending their battery life). With MK one can construct an entire keyboard of frequently used synthetic functions, speeding the construction of heavily synthetic programs.

MK requires three inputs for each key assignment. The first two inputs are decimal codes from the byte table, while the third input is the user keycode in the same notation that ASN uses (row, column, negative for shifted key). In the interest of speeding execution, MK does not pack the key assignment registers.

Example 1: Assign RCL b to the SIN key. Begin with XEQ MK . If you get the message "RESIZE > = 12", then increase the SIZE to at least 012 and R/S. After the program checks the key assignment registers it displays the prompt "PRE↑POST↑KEY". Key in 144 ENTER↑ 124 ENTER↑ 23. The RCL prefix is decimal 144 (see row 9 column 0 of the byte table). The postfix b is decimal 124 (hex 7C). The third entry is the keycode (row 2 column 3 unshifted). Press R/S to make the key assignment. When the "PRE↑POST↑KEY" prompt comes back, the assignment is made (and you can hold it down to see the previous XROM 01, 60).

Example 2: Continuing Example 1, let's assign STO b to the SIN^{-1} key. Assuming that you haven't moved the program pointer from the end of Example 1, you can enter the three inputs and proceed. But first let's illustrate the key-checking capability of MK . Key in 145 ↑124↑23 (STO b to row 2 column 3 unshifted) and R/S. You'll get the message "KEY TAKEN", then the prompt "KEYCODE?". Key in -23 (row 2 column 3 shifted) and R/S. When the prompt for the next key assignment comes back you're done. Hold down the SIN^{-1} key to see the preview XROM 05, 60.

COMPLETE INSTRUCTIONS FOR MK

1. SIZE 012 (at least) is required. If the SIZE is not large enough, you'll get a message to resize, after which you can R/S.

2. Execute MK . The program will check the key-assignment registers from the bottom up, eventually pausing to display the number of free registers available for making key assignments, to the nearest ½ register. Doubling this number will give the number of additional assignments for which there is room.

3. If you get the message "NO ROOM", you can delete some key assignments and/or reduce the size (but not below 12), then push R/S, which will restart the program and initiate another register count. Or you can delete some programs, or execute the ROM routine PK to pack the key-assignment registers, then execute MK again. (Just deleting a key assignment will not necessarily give more room unless the PK routine is also executed.)

4. After pausing to display the number of free registers, the program will stop with the prompt "PRE↑POST↑KEY". Key in the decimal equivalent of the first byte (prefix), ENTER↑, the second byte (postfix, ENTER↑ and the keycode (notation for keycode same as in the display of normal key assignments). Then push R/S. (For example, 159 ENTER↑ 26 ENTER↑ -81 R/S will assign Tone 26 to the shifted key in row 8, column 1.) If the key assignment has been successfully made, the program will prompt for the next key assignment. (No count is made of the assignment number.) The stack is cleared before each such prompt, so that for example if only a postfix and keycode are entered, the prefix will be taken to be zero.

5. Entering zero for the keycode (or just pushing R/S with no entry) will give a display of the number of free registers remaining (pausing), followed by a re-prompt for the key assignment.

6. When you are finished making assignments, simply leave the program by going wherever else you wish to go in program memory. There is no "termination procedure" necessary in this program, and there is no need to worry about whether you have made an even or odd number of assignments.

7. The message "KEY TAKEN" followed by the prompt "KEYCODE?" means that there was already an assignment to that key. You may then either delete the already-existing assignment and push R/S to make the new assignment, or else enter a new keycode and push R/S. "NO SUCH KEY" followed by "KEYCODE?" means that you tried to assign a function to a nonexistent key. Enter a new keycode and push R/S.

After either error message, the keycode originally attempted can be found in the X-register, so pushing R/S amounts to entering the same keycode again. Whether or not a new keycode is entered, the rest of the stack is irrelevant, as the original prefix and postfix are remembered and reused after the keycode is entered. However, if zero is entered after the "KEYCODE?" prompt, then the original prefix and postfix are forgotten, and after pausing to display the number of free registers the program stops with the "PRE↑POST↑KEY" prompt for a fresh assignment.

8. The message "DONE, NO MORE" means that the last assignment has been successfully made, but there is no more room for further assignments. If you still want to make more assignments, proceed as in the "NO ROOM" case (Step 3).

9. After any stop due to an error message (Steps 3, 7, or 8), or if the calculator has been turned off and on, the program recounts the key-assignment registers to ensure that new assignments are stored without any overlap or gaps. (The need for a register recount is decided by testing Flag 20; if that flag is clear then the registers are recounted.)

10. Using PK after MK will maximize the number of available program registers and also give you a count of the total number of assignment registers used. This is especially valuable if a status card is to be recorded.

WARNING: (a) If you must pack or change the size in between key assignments, turn the calculator off and on (or clear Flag 20) to signal the program to do a register recount. (b) Don't store anything in registers 09 10, 11, or change Flags 07, 09, 10, or 20 between key assignments. As with LB , the only data registers used are 06-11. R_{06}, R_{07} and R_{08} contain the prefix, postfix, and keycode for the last-entered assignment; R_{09} contains the index for indirect storage of the next key assignment and R_{10} contains the c-register contents for the lowered curtain, both as in the LB program; R_{11} holds (when Flag 10 is set) the first of a pair of key assignments. Flags 08, 09, 10, and 20 are used by the program for internal control.

MORE EXAMPLES OF MK

Example 3: Assign an F7 byte jumper to the X>Y? key. XEQ MK and when the input prompt appears, key in 247 ENTER↑ 142 ENTER↑ -71 R/S. The 247 is hex F7, the 142 corresponds to PROMPT, and -71 is the keycode for the x>y? location. If the x>y? location is assigned you'll get the "KEY TAKEN" message. In that case you can manually clear the assignment (ASN ALPHA ALPHA SHIFT x>y?) and R/S. If you do not have a card reader plugged in, this F7 byte jumper previews as XROM 30, 14. If a card reader is attached the preview is 7DSP2, which is the real XROM 30, 14. The byte jumper will not execute as 7DSP2 despite the misleading preview.

Example 4: Another handy use of MK is the assignment of normal (non-synthetic) HP-41C functions. For example, decimal entries 168 and 14 for prefix and postfix will give SF 14 for faster recording of clipped cards. Assigning your favorite two-byte functions can save time when keying up a long program.

Example 5: XROM labels are assignable using ASN, with one exception--the XROM name. For example, the card reader header is XROM 30, 00 (decimal 167, 128) and the PPC ROM header is XROM 10,00 (decimal 162, 128). The decimal codes are calculated using XL . XROM 10, 00 can be used in a program (created via MK or LB) to detect the presence or absence of the PPC ROM. If the PPC ROM is present it appears to have no effect on the status of the calculator, while if the PPC ROM is absent it gives NONEXISTENT. Therefore, it is tentatively suggested that routines requiring the PPC ROM begin with the steps

```
CF 25
XROM 10,00   (decimal 162, 128)
```

BACKGROUND FOR MK

A. The Storage of Key Assignments

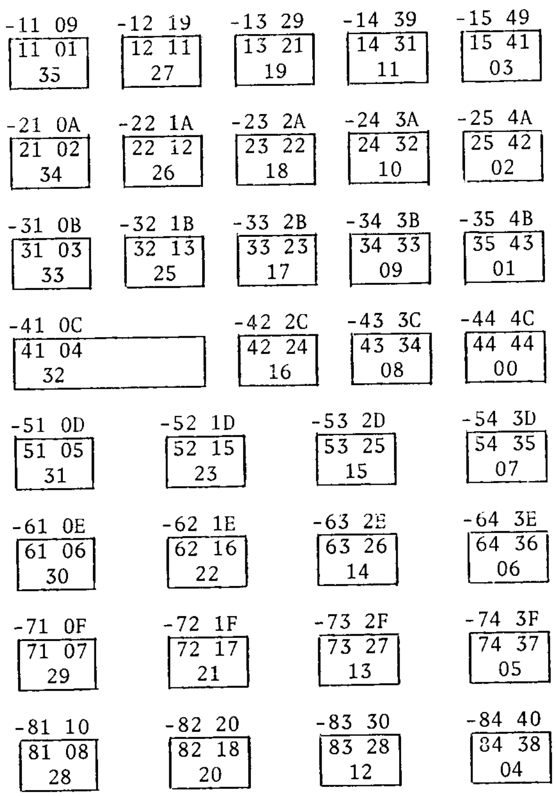
Key assignments made by the HP-41C's ASN function fall into two basic types: function assignments and global label assignments. In either case, when a key assignment is made there are two basic operations involved: (1) changing a bit or "flag" from 0 to 1 in the appropriate status register (the "⊦-register" for unshifted-key assignments and the "e-register" for shifted-key assignments) to inform the calculator that the key has an assignment made to it, and (2) storing information on what function or global label is assigned to the key.

In the case of global-label assignments (which actually applies only to the Catalog 1 labels in RAM), a code for the key is stored within the LBL instruction itself. In the case of function assignments (which applies to everything in Catalogs 2 and 3, including ROM programs described by global labels such as the ones in this ROM) the functions and key codes are stored in the registers at the bottom of program memory--the registers with absolute addresses 192 (decimal) on up--at the rate of two function assignments per register. (See V6N6P19 of the *PPC CALCULATOR JOURNAL*, for a discussion of the HP-41C's memory structure.) Between the uppermost of these "key-assignment registers" and the lowermost program register (that containing the final .END.) lies a block of unused or "free" registers into which either programs or key assignments may expand; the number of registers in this free block is shown in the 00 REG nn or .END. REG nn display (and can also be calculated using the number-of-free-registers ROM routine F? -- see the description of that routine for details.)

In this description we shall be mainly concerned with function key assignments and how they make use of the key-assignment registers.

The way in which function key assignments are stored in a register is as follows: The first (leftmost) byte is always F0 (unless otherwise specified, byte values will always be given in hexadecimal in this Background), which the calculator uses to identify the register as containing key assignments, and the other six bytes are divided into two groups of three bytes each. Each three-byte group (which we shall call a "half-register" for simplicity) contains the complete description of a function key assignment: the first two bytes describe the function, while the last byte describes the key to which the function is assigned according to an "internal keycode" which is not the same as the familiar "user keycode"(11 for the upper-left key, etc.). Fig. 1 shows the relation between the user keycode, the internal keycode, and the bit number in the ⊦ or e-register for each key. (Notice that 00 does not occur among the internal keycodes--it is reserved for indicating "void" half-registers, as we shall see shortly.)

Since the most general "function" that can be assigned to a key is described by the first two bytes of the 3-byte "half-register", there are 256 x 256 or 65,536 assignable functions to each key. Only a small fraction of these are actually used in normal HP-supported key assignments, however; these are (1) "one-byte" functions in which the first byte is a "filler" byte 04 (actually 00 through 0F seem to serve the same purpose) and the second byte describes the function itself, which can be any of the mainframe functions listed in Catalog 3, and (2) "two-byte" functions which include all of the peripheral (XROM) instructions, the first byte being A0 through A7 and the second byte being (in principle) arbitrary. (See the XL routine description for the relation between byte values and XROM numbers.) The rest of the 65,536 functions cannot be assigned using the usual ASN operation, but must be "synthesized" by constructing the string of bytes and storing it in the key-assignment register, remembering also to set the appropriate bit in the appropriate status register. All of this is done by the key-assignment program MK and its programmable versions 1K and +K . The behavior of these "synthetic" key assignments when the assigned keys are pushed is sometimes, but often not, the same as if the bytes appeared as instructions in program memory, and an even different behavior may result when the keys are pushed in PRGM mode. Many synthetic key assignments have been found to be extremely useful in programming and operating the HP-41C, while many others still remain to be explored. Examples will be given in the instructions for MK and elsewhere; in this section we shall be more concerned with the manner in which the assignments are stored and accessed in the key-assignment registers.



On each key appears the user keycode followed by the internal (hex) keycode for the unshifted key. Above the key appears the same information for the shifted key. At the bottom of the key appears the decimal number (using flag notation) of the bit which identifies the unshifted key in the ⊦-register and the shifted key in the e-register. (See articles by W. Wickes (3735), G. Istok (2525), and W. Kolb (265) in *PPC CALCULATOR JOURNAL,* V6N7P32d and V7N2P29c, P30-34, and P36-37.)

Fig. 1. Keycodes and Bit Numbers Used in Key Assignments

Fig. 2 shows an example of a key-assignment register and the teminology we are using here. If the third byte (the internal keycode) in any half-register is 00, then that half-register is considered not to have any assignment (regardless of the first two bytes) and will be called a "void" half-register. Whenever a function key assignment is deleted using the normal methods (ASN ALPHA ALPHA ...), then the corresponding bit in the ⊦- or e-register is set to 0, the key assignment registers are searched for the assignment (see "searching procedure" below), and the internal keycode change to 00 *without altering* the other bytes in that half-register. Thus, if the assignment to key -15 is deleted in the example of Fig. 2, the byte 01 will be changed to 00 (making the right half-register "void"), but nothing else in the register will be changed.

The seven bytes below show the contents of a register containing the assignment of the function MEAN (hex code 7C; note the filler byte 04) to the upper-left key (user keycode 11; internal keycode 01), and the card-reader function VER (XROM 30, 05; hex code A7 85) to the shifted upper-right key (user keycode -15; internal keycode 49). Bit 35 in the ⊦-register and bit 03 in the e-register would also be set to 1 to signal the existence of these key assignments.

```
F0  A7  85  49   04  7C  01
     "Left        "Right
      half-        half-
    register"    register"
```

Fig. 2. Example of the Storage of Key Assignments in a Register

When a half-register becomes void, the calculator does *not* make any attempt to "pack" the key-assignment registers to eliminate the void, even if a PACK command is given, with the following exception: If *both* halves of a register are void, then the calculator will "delete" the register, moving down the assignments above it to fill the gap, but this deleting is not done until either (1) packing of program memory occurs, or (2) the calculator is turned off and on. At no time does the calculator ever perform any "horizontal" shifting of half-registers to eliminate voids. Thus, deleting key assignments will not increase the number of free registers unless two voids happen to appear in a single register, and even then not until the machine is turned off or a packing performed. But (synthetic programming to the rescue!) the ROM routine PK can be used to overcome this problem; when executed it performs the necessary shifting of half-registers and eliminates all voids (except of course for one if the number of function assignments is odd). See PK for more details on that routine

The fact that an assignment register with two void halves is (at least temporarily) preserved can be exploited to make a "Clear Function Assignments" status card (see *PPC CALCULATOR JOURNAL,* V7N8P22a). Starting with no key assignments, make one assignment and then delete it, and without turning the machine off write a status card, keeping only Track 2. To use the card at any later time, just run through Track 2 and turn the machine off and on--which will clear the prompt for Track 1 and leave you with no function assignments. (The global-label assignments will still remain.)

Consider next the "searching procedure" for function key assignments. When the calculator has to search the assignment registers for a given keycode (to execute an assigned function, to delete an assignment, or locate voids), the search order is from the lowest register on up (i.e. absolute addresses 192, 193, 194, ...) and from right to left within a register. The search ends when a half-register with the desired keycode is found, or when a non-key-assignment register is encountered (such as one with all 0's), or when the .END. address is reached, whichever comes first.

In general, when a key is pressed in USER mode, the calculator first looks at the ⊦- or e-register to see if the appropriate bit is 1. If so, the function key assignment registers are searched as described in the above paragraph. If the keycode is not found there, then the global labels are searched starting with the last one in Catalog 1 and working toward the beginning, and if the keycode is still not found a default function is assumed (ABS, 1/x, CAT, or temporary fade-out).

On the other hand, if the bit in the status register was 0, then if the key is A-J or a-e the machine will search for a corresponding local label in the program where the pointer is. If the label is not found, or if the key was not A-J or a-e, then the normal machine function is executed or loaded into program memory. (It is the search for the local label that accounts for the considerable time delay that can occur when, say, X<>Y or R↓ keys are pushed in USER mode with the program pointer in a long program. The annoying delay can be easily eliminated by assigning these functions to their own keys, e.g. assign "X<>Y" to the X<>Y key, since key assignments take precedence over local-label searching and are therefore executed very quickly.)

The above search procedures, incidentally, were found by making several different assignments to the same key and seeing which one the machine actually found. Multiple function assignments to the same key can be made by copying program MK into RAM and executing it after first deleting lines 113 and 114 which check for already-used keys. One can also make multiple function or global-label assignments by setting the bit in the status register back to 0 between assignments (such as by storing 0 into the appropriate status register) and using the normal machine methods to make the assignments to the same key. The search order is really of no consequence to the user except as a matter of curiosity. It can, however, affect the way in which function assignments get stored in the assignment registers, which we shall now describe.

When the ASN operation is used to make a function key assignment, the bit in the appropriate status register is checked. If it was already 1, indicating that the key was already being used, the internal keycode is searched for among the assignment registers and the global labels (using the above searching procedures) and changed to 00 to delete the old assignment. Then, to make the new function assignment, the assignment registers are searched (again by the register searching procedure) for the first void half-register (i.e. the first occurrence of 00 as a keycode), and if one is found, the new assignment is stored there. If no void half-register is found, then all of the contents of the key-assignment registers are moved up one register (in the manner of a stack lift--if there is no room the whole procedure is aborted) and the new assignment stored in the right half of the bottom register (192), the left half being 00 00 00. (The next function assignment would, following the rules just described, then be stored in the left half of this register.)

The placement of function key assignments made synthetically by program MK (or its programmable versions +K and 1K) is somewhat different. In the interest of saving time, the key-assignment registers are not searched for voids, but simply counted in order to locate the lowest free register. (The counting is actually done by routine LF which the key-assignment program calls as a subroutine.) If the uppermost used key-assignment register has a void right half, the first assignment made by MK will be stored there; otherwise the left-half of the next higher register will be used. As assignments are made by MK the order of storage is from left to right within a register, and the registers are filled in ascending order.

Fig. 3 summarizes the searching procedure and the storage order using the normal ASN and the MK methods of making assignments. We have also given an example of both methods used together and, for completeness, the effect of the key-assignment packing program PK.

(a) The order in which assignment half-registers are searched by the HP-41C.

(b) The result of making seven function key assignments using the machine's ASN function. The half-registers are numbered in the order in which the assignments were made.

(c) The result of making seven function key assignments using MK (or 1K and +K).

(d) The result of making assignments 1-5 using ASN, deleting assignment 2, then making assignments 6-9 using MK and finally deleting assignment 6.

(e) The result of making the assignments as in (d) and then executing PK. (See PK for more details.)

* indicates a void half-register in which *all three* bytes are zero.

(a)

Absolute Address (decimal)		etc.	
195	F0	8	7
194	F0	6	5
193	F0	4	3
192	F0	2	1

(b)

F0	2	1
F0	4	3
F0	6	5
F0	Void*	7

(c)

F0	7	Void*
F0	5	6
F0	3	4
F0	1	2

(d)

F0	8	9
F0	Void	7
F0	Void	1
F0	4	3
F0	Void*	5

(e)

F0	7	Void*
F0	8	9
F0	5	1
F0	4	3

Fig. 3. Examples of the Storage of Function Key Assignments Created by Various Methods

The above described procedures by which the 41C searches for and makes assignments when ASN function is used really should be termed "effective" in the sense that they were ascertained by studying the effects of making various combinations of assignments and deletions and examining the assignment registers. The actual microcode may not necessarily perform these steps exactly as indicated here. For those who wish to investigate further, a program "PRK" which prints out the contents of the key assignment registers can be constructed by making a few modifications to the LF routine; see LF for a description of the "PRK" program.

A few remarks are in order concerning the use and counting of the "free" registers between the key-assignments and the final .END. Both the routine LF (Locate Free Register Block, called by MK, 1K, and +K as a subroutine) and PK (Pack Key Assignment) start with register 192 in absolute decimal address and continue recalling and processing registers until an empty register is encountered (unless the .END. is reached first). The registers from this empty register on up to just below the .END. are also assumed to be empty. If, on the other hand, these supposedly free registers should happen to contain data stored intentionally or unintentionally by the user, or if some future HP peripheral should make use of these registers, then problems could arise. Both **LF** and **PK** should work properly as long as there is at least one empty register immediately above the key assignments, though repeated assigning of keys by **MK** or its programmable versions might cause key assignments to wipe out the data. But if the block of data is immediately adjacent to the block of key assignment registers, then **LF** will run through and process both blocks as if they were one large block of key assignments, changing the initial byte of each register to F0 before restoring. The result could be spurious function assignments, not to mention drastic changes in the data. The 41C itself seems to have problems of its own when data is stored between the key assignments and the .END. With no empty register above the key assignments, the calculator sometimes (depending on the data) appears to get into an infinite loop upon packing or turn-on (momentarily removing the batteries will get out of this loop). With an empty register in-between, the infinite loop does not occur, but the 41C only shows in the "REG" display, and allows you to use, those empty registers between the .END. and the data. When the key-assignment-register stack is "lifted" by the addition of new assignments using ASN, the spruious data is also lifted, further decreasing the number of registers free--yet the data block is not lowered when assignments are deleted, even when registers with two voids are eliminated. The result is a gradual shrinking of usable program memory. The clear-key-assignments routine **CK** can be used to remedy the situation; it simply wipes out everything up to the .END., spurious data and all.

B. Synthetic Key Assignments

Like normal key assignments, synthetic key assignments fall into two categories: one-byte and two-byte. This terminology refers to the coding of the assignment in the key assignment registers and is not necessarily related to how many bytes actually get loaded into program memory if the assigned key is pushed in PRGM mode.

1. One-byte key assignments are of the form On ab when n, a and b are hex digits. The byte On is the "filler byte"; the digit n appears to be ignored by the system as far as identifying and executing the function is concerned. When a normal assignment is made using ASN, n is set equal to 4. The byte ab, then, completely describes the function (hence the term "one-byte"). An example of a normal one-byte assignment is the assignment 04 7C in the "right-half-register" of Fig. 2; byte 7C is the hex code for the MEAN function.

To create any one-byte key assignment, execute **MK** and, after the PRE↑POST↑KEY prompt, key in (in decimal) the filler byte (0 to 15), ENTER↑, the function byte, ENTER↑, the keycode, and R/S. (Incidentally, **MK** clears the stack before prompting for an assignment, so you need not enter the filler byte if you want it to be zero. As far as is presently known, the choice of filler byte is a matter of the user's taste.) For example, 4 ENTER↑ 124 ENTER↑ 11 will create the MEAN assignment shown in Fig. 2, although, it could of course also be created using ASN.

The relation between most of the normal one-byte assignments and their hex and decimal codes are shown in the Hex/Decimal Byte Table. Normal one-byte assignable functions shown in this table, start with Row 4 (+, -, *, /, etc.) and continue through Row 9 (RCL, STO, ..., TONE). They include functions A8 through AD (SF, ..., FC?) plus a few isolated others: CO (END), CE (X<>), CF (LBL), DO (GTO), and EO (XEQ). This accounts for 107 of the 117 assignable functions listed in Catalog 3. What about the 10 others?

They are the non-programmable functions like SIZE, etc, not shown in the Byte Table, beacuse that table shows only the functions represented by the bytes when they are loaded into program memory and there is no byte which, when loaded into program memory gives the SIZE function! It turns out, however, that the bytes representing all ten of the non-programmable functions lie in Row 0 of the Byte Table, the row containing the null byte and the one-byte labels 0-14. Byte 06, for example, means LBL 05 in program memory, but if a key with the one-byte assignment 06 is pressed (in PRGM mode or otherwise) we do not get a LBL 05, but instead SIZE_ _ _. The bytes representing the 10 non-programmable normal function key assignments are 00 (CAT), 02 (DEL), 03 (COPY), 04(CLP), 06 (SIZE), 07 BST), 08 (SST), 09 (ON), 0A (PACK), and 0F (ASN).

There are 256 possible values that a byte can have, and after subtracting the 117 normal assignable functions of Catalog 3, we are left with 139 one-byte key assignments , which cannot be made with the machine's ASN function, but which can be made using **MK** . All of these 139 synthetic one-byte key assignments show strange "function names" when their assigned keys are pressed, or in a PRKEYS listing, presumably characters pulled out of sections of microcode which were never intended to contain function names. (The longest named "function" is the assignment with hex byte 25, which has the distinguished title of)@@αd@"%(@@Γc !$1 @@ᵀbDCAB@@♦aα when printed out, but appears only as ᵀbDCAB@@⁻a in the display.) Some characters which show up in the printed names appear as blanks in the display. The names cannot always be used to uniquely identify the functions, as several synthetic assignments, for example, share the name ≠αe2. Many of the names are displayed with prompt underlines, and the behavior of the function may or may not depend on how the prompts are filled in. As strange as the names are,some of the one-byte synthetic key assignments have proven to be extremely useful as programming aids. A few of the more useful or otherwise noteworthy one-byte synthetic assignments follow. First the hex code of the second byte is given with the decimal equivalent in parentheses (remember that the first byte, the filler, can be any number from 0 through 15 decimal). After that the function "name" is given; boxed-in characters show up normally on the printer, but as blanks on the 41C display.

05(5): @N% in PRKEYS listing; not displayed when key is pressed. Acts like R/S as long as program is not running (i.e., starts execution if in non-PRGM mode, enters STOP instruction if in PRGM mode).

0B(11): @J in PRKEYS listing; not displayed when key is pressed. Deletes current program line regardless of whether calculator is in non-PRGM or PRGM mode. (As with the backarrow key, the program pointer is then positioned at the previous line with the line number decreased by one. However, if the current line's number had not been calculated, as is the case, just after running a program, then the previous line will be numbered 4094.) NOTE: If this function is assigned to a shifted key, the "shift" is not cleared after the key is pushed.

0C(12): @+ in PRKEYS listing, not displayed when key is pressed. Acts like the ALPHA, PRGM, or USER key depending on which row the key is in (see W. Wickes (3735) *SYNTHETIC PROGRAMMING ON THE HP-41C,* reference below, p.21).

0E(14): @; in PRKEYS listing, not displayed when key is pressed. Acts like shift key.

10(16): B One of the "Q-loader functions". In non-PRGM mode it simply enters zero into X. But in PRGM mode it enters two program lines. The first is a "0" digit-entry line without the "separator" null that pushing the 0 key would produce. It can be used to merge a 0 with a previous digit entry line if desired. The second is a text line whose characters are,in reverse order, the bytes in the Q-register. The Q-register becomes cleared afterward. Any leading null bytes in Q (which would appear as trailing nulls in the text line) are omitted. By storing the proper number in the Q-register (see "two-byte key assignments") and using the Q-loader, one can quickly generate any text line of seven characters or less not ending in a null. See W. Wickes' (3735) "Synthetic Programming" book for more details on the use of the Q-loader.

11 (17):	axBde^T>52-$T+XμIAα_ _ _ (first 10 characters not shown in display)
12 (18):	B_ _
13 (19):	μΓ
14 (20):	DCAB@@♦aα (♦ becomes ¯ in display)
15 (21):	X
16 (22):	H⊥D
17 (23):	Aα_ (requires alpha filling of prompt)
18 (24):	≠αē2_ _
19 (25):	OD
1A (26):	-
1B (27):	Hb_ _
1C (28):	≠αē2_ _

These functions are also Q-loaders, acting like 10 except that the digit-entry line 0 is replaced by the digits 1 through 9, the decimal point, E, and the negative sign. (Executing the "negative sign instruction", incidentally, puts zero in X, as opposed to the ordinary minus instruction, which looks the same in a program listing). For those giving numerical prompts, the result seems to be independent of what number is filled into answer the prompt. An exception to all of this is function 17, which requires an alpha entry to fill the prompt and this alpha entry (rather than the original Q-register) ends up in the text line. All of the others are equally suitable for creating text lines, though the non-prompting ones are the most convenient to use. Function 1B provides a quick way to product the E instruction (which acts like 1, but is faster), but an even quicker way is provided by a two-byte synthetic key assignment (see below) which behaves the same, but does not require a prompt to be filled.

1D (29):	@♦AHHH
1E (30):	@NQ
1F (31):	≠αe2_ _

These are the "GTO, XEQ, and W Q-loaders" respectively. In PRGM mode they create the instructions GTO ^{T}..., XEQ ^{T}..., and W^T..., where the text portion is the contents of the Q-register in reverse order, excluding leading nulls. The W^T_ function does not seem to do anything besides (usually) lock up the calculator.

A7 (167): The celebrated eGØBEEP_ _
This was one of the first useful one-byte synthetic functions to be explored (Robert W. Edelen (339) *PPC CALCULATOR JOURNAL,* V7N3P16). When the prompt is filled by any two-digit number, n, the result is the function XROM 28, n if n is less than 64, and XROM 29, (n-64) if n is between 64 and 99 inclusive. Since XROM 29 is the printer, whose instructions run from XROM 29, 01 to 29, 24, we can create all of the printer instructions from the one eGØBEEP key by filling in the numbers 65 through 88--even if the printer is not plugged in. (The functions will not execute or display their alpha names of course, until the printer is plugged in.) The eGØBEEP function doubles as an alpha LBL instruction if the prompt is filled by an alpha entry, which saves pressing the shift key for LBL if eGØBEEP has already been assigned to an unshifted key.

CD (205): a_ _
This is the "LBL Q-loader". When the prompt is filled (the value being irrelevant), the result is LBL^T... with the text taken from the Q-register as in the above mentioned Q-loaders.

F0 (240): W
This is one version of the famous "byte jumper" discovered by William C. Wickes (see *PPC CALCULATOR JOURNAL,* V7N4P26). When executed in non-PRGM mode it casuses the program pointer to jump forward in program memory by a number of bytes equal to the second hex digit of the byte at which the pointer was originally positioned. (The last hex digit preceeding the instruction that is being desplayed in PRGM mode). The bytes jumped over are replicated in the alpha register, as if a text instruction had been executed. (The bytes in program memory remain unaltered.) Byte jumping allows one to enter the middle of a multi-byte instruction (where he can alter it if desired), and also allows one to examine the byte structure of program lines by decoding the results in the alpha register. Many ways of using the byte jumper have appeared in the *PPC CALCULATOR JOURNAL,* for example see V7N6P43 and V7N10P20. Also read W. Wickes (3735) "Synthetic Programming" book. Incidentally, if this particular version of the byte jumper is inserted into program memory in PRGM mode, the result is merely GTO ØØ, but other versions have more interesting uses in PRGM mode (see the two-byte assignments below).

F1 through FF (241 through 255): These give other versions of the byte jumper, acting the same as F0 in non-PRGM mode. Their behavior in PRGM mode varies, depending in some cases on the numbers filling the prompts.

These are only a few of the one-byte synthetic functions; many of the others still need to be explored. Program MK and its programmable versions should facilitate this exploration. (See for example application program using +K which makes a whole row of one-byte key assignments at once.) Incidentally, the wand provides another means of exploration, as the 2-byte paper-keyboard barcode S4 ab (when S is the proper checksum) when scanned appears to produce the same results as the one-byte key assignment 0n, ab, with some exceptions in row 0 of the hex table (a=0). For example, the barcode S4 10 gives the 10 Q-loader, and 54 F0 gives the F0 byte jumper.

2. Two-byte key assignments are of the form ab cd where a≠0. The only normal assignments of this type are of the case a=A and b=0, 1, ..., 7, giving the XROM functions. See **XL** for a discussion of the relation between XROM numbers and byte values; the results can be summarized by saying that if we combine the three hexadecimal digits b, c, d into one 12-bit number and divide that into two 6-bit groups, then when converted to decimal the two 6-bit numbers are the XROM numbers of the functions. (For example, A7, 81 XROM 30, 01, the card-reader function MRG, because 781 hex = 0111,1000,0001 binary=011110,000001 regrouped binary=30,01 decimal.)

Any two-byte key assignments with a≠A and/or b>7 must be created synthetically. Regardless of what the function turns out to be when executed or loaded into program memory, the name displayed when the key is held down (or appearing in a PRKEYS listing) is always an XROM number, which we call a "pseudo-XROM number" because they do not actually refer to any function in any external ROM. The pseudo-XROM numbers are related to the digits b, c, d of the two-byte key assignment ab cd in exactly the same way as described above for normal XROM functions, the digit a being ignored in figuring the pseudo-XROM number. (Apparently when the calculator observes that a≠0, it assumes withoug checking that a=A, and proceeds to display the XROM number based on b, c, and d. When the function is executed or loaded into a program, however, all four digits are taken into account.) For example, the synthetic assignment 90 7C (RCL b, as we shall see later) displays temporarily as XROM 01,60, because 07C hex=0000,0111,1100 binary=000001,111100 regrouped binary=01,60 decimal. The assignments 107C, 207C, ..., F07C will also display as XROM 01,60--including , of course, the real XROM function A0 7C which actually accesses function #60 of XROM #01. It is evident that, although the names displayed by the two-byte assignments are much more systematic than these displayed by one-byte assignments, they are still far from being unique.

Two-byte key assignments fall into several behavioral types, perhaps best illustrated with examples.

a. In general, if the first of the two bytes represents by itself a one-byte function, then the key assignment tends to take on that function when executed or loaded into program memory, the second byte being ignored. Thus, the assignment 40 41 (hex bytes) is equivalent to the + function (the normal assignment of which would be 04 40) since 40 by itself represents the complete function +.

b. The above also seems to apply to some synthetic one-byte functions. For example, 1M cd seems to behave like 00 1M, the Q-loaders (M=0 thru F), regardless of the second byte cd--with the exception that the name temporarily displayed is the pseudo-XROM number in the two-byte case. The assignment 1B 00 thus produces (in PRGM mode) the E digit-entry line along with the text line from Q, just like the assignment 00 1B, but without the need to fill in a meaningless prompt. The two-byte assignment Fn, cd also gives the byte jumper when executed in non-PRGM mode, but produces completly different results in PRGM mode (see below).

c. If the first byte ab corresponds to the prefix of a normal two-byte function, (90-9F, A8-AE, CE-CF), then the second byte is interpreted as the postfix and the complete two-byte nstruction is executed or loaded into program memory. This has two useful consequences:

(1) A complete instruction, normally requiring the filling of prompts, can be executed by a single key stroke. For example, assign the two byte function A8 0C to key 11 (decimal inputs 168, 12, 11 in **MK**) pushing key 11 in user mode will then execute SF12 or load that instruction into program memory. (A8=SF; 0C=12). Similarly, AE 73 (decimal 174, 115) will give GTO IND X, and 92 F3 (decimal 146, 242) will give ST+IND Y.

(2) Of greater significance, by choosing appropriate values for the second byte we can directly access registers heretofor impossible and create non-standard labelsand display modes. Consider for example, an assignment whose first byte is 90 (STO), 91 (RCL) or any other register-involving operation. The normal postfixes allowable by the 41C are from 00 to 99 decimal, or 00 to 63 hexadecimal, and also from 70 through 74 hexadecimal for accessing the T, Z, Y, X, and L registers (see the byte table), along with similar bytes in the second half of the byte table for indirect operations. If we synthesize instructions with other postfixes we find that 64 through 6F access registers 100 through 111, while 75 through 7F access a whole new set of registers which along with the stack and last X form the 16 "status registers". For example, 90 7C recalls from a register which has come to be called the "b-register" because when entered into a program (or printed while being executed with the printer in NORMAL or TRACE mode) the instruction shows itself as RCL b. See the Byte Table for names of the other status registers (some of which appear differently on the printer than in the display), and see for the functions of these registers. For the results of using "illegal" postfixes with the TONE instruction, see the tone table accompanying the **TN** routine description.

d. Assignments of the form, **BM** cd seem to act like compiled (or uncompiled, if cd=00) GTO's when executed in non-PRGM mode, and may be useful for moving the program pointer around. Assignments with first byte in rows D and E also act sometimes as a compiled and sometimes as a label-searching GTO or XEQ.

e. Of particular interest are assignments of the form Fn cd, wich act as the ordinary byte jumper in non-PRGM mode, but in PRGM mode because what has been called the "byte masker" or "byte grabber". If the key with assignment Fn cd is pushed in PRGM (and of course USER) mode, there will be inserted into program memory the three-byte sequence Fn ØØ cd (provided that cd>0E; otherwise we just get a GTO instruction). As with a normal three-byte instruction, the calculator will open up a register of 7 nulls prior to the insertion if and only if there were less than three nulls in which to insert the "instruction". Once the instruction is inserted, however, the machine will treat it as a text line which (if n>2) will not only include the 00 and cd bytes, but will absorb the following n-2 bytes, thus "masking" or "grabbing" bytes from the following instruction.

LINE BY LINE ANALYSIS OF MK

We begin MK by clearing flag 07 to indicate the prompting version, setting flag 09 to ensure that the number of free registers will be displayed, and checking that the size is at least 12 (lines 02-07). Then the assignment registers are counted by LF, the index bbb.eee for the free block ending up in X. This index is stored in R09, the c-register contents for the lowered curtain (which LF left in Y) stored in R_{10}, and the first assignment of the topmost assignment register (which is contained in the first 6 characters of alpha) ASTO'd in R_{11} (lines 10, 13-15). Lines 11-12 and 16-17 determine whether there is no room for any more assignments by branching to LBL 07 if bbb>eee. If everything is okay, we set flag 20, (line 18) to indicate that the registers have been checked, and then proceed to display the number of free registers, since flag 09 was set and line 20 is bypassed. If the program has to do another register check later, however, or if versions 1K or +K are executed, then flag 09 will be clear at this point and line 20 will be executed, bypassing the number-of-free-registers display and the prompt for input.

The number of free registers is found and displayed in lines 22-31. The calculation is actually done in a short subroutine starting with LBL 11 (lines 197-209) which is shared with routine F? (see that routine for details). The result is to the nearest ½ register and is displayed in FIX 1 format while the original display mode is preserved by recalling and restoring the flag (d) register (lines 24-29). After pausing with this display the key assignment inputs are prompted for (lines 32-37) and stored in R_{06}, R_{07}, and R_{08} (lines 38, 43-48). Flag 09 is cleared for future use (line 49), and in lines 50-54 a branch back to LBL 01 is made for another assignment register check if either flag 20 is clear (indicating that the machine may have been turned off since the last register check, which could alter the assignment registers) or R_{10} contains non-alpha data (indicating that the registers have been tampered with). This is an attempt (though not foolproof) to prevent disaster in case the user warnings outlined in the instructions are ignored. If all is okay we proceed to the processing of the input key code, starting at LBL 13 (line 55).

At this point let us look at the operation of the programmable versions 1K and CK. Routine 1K starts at line 39 clearing flag 20 and setting flag 07 (to signal the need for a register check and set the non-prompting version). After the inputs are stored and flag 09 cleared,the "No" response to line 52 causes a branch to LBL 01 (line 08), where upon an assignment register check along with the initialization of R_{09}-R_{11} takes place as with MK. Then since flag 09 is clear, lines 19-20 branch to LBL 13 (line 55) to start processing the input keycode. Routine +K starts at line 42 and is almost identical to 1K except that flags 20 and 07 are not changed. Unless flag 20 has been cleared or R_{10}'s contents altered, the branch to the assignment register check (line 54) will be bypassed, thus saving time. Since flag 07 is not changed by +K, this routine may be used to continue assignments with either the prompting or non-prompting version. Flag 07, incidentally, is checked in line 154 after the key assignment has been made and stored to decide whether to simply RTN or prompt for another assignment. Returning to the processing of the keycode starting with LBL 13 (line 55), the first action is to discard any fractional part and decide what to do if the keycode is zero. The pile of conditionals in lines 58-61 has the following effect: If the keycode is non-zero or flag 20 is supposed to be set by now, line 61 is bypassed and we go on to further processing in line 62. On the other hand, if the keycode is zero and flag 07 is clear (or if flag 20 somehow got cleared, which cannot happen unless the program has been interrupted) then line 61 is executed and we go back to LBL 02 for a number-of-free registers display. Thus, entering zero for a keycode will give a display of the number of free registers and a reprompt, but only in the prompting version; in the non-prompting version a zero keycode is interpreted as an error farther down in the program. A further result of all this is that by the time line 62 is reached, flag 20 has been cleared; it will be set again later (line 153) if there is room for future assignments.

Lines 62-101 convert the user keycode (which remains in R_{08}) to the internal keycode and also calculate the number of the bit to be set in the ⊦ - or e-register (see Fig. 1 of "Background for MK "). Flag 09 (which is always clear at the beginning of this sequence) is set in lines 62-63 in the case of a shifted key. Basically, if MN represents the digits of the absolute value of the user keycode (i.e., row M, column N), then the hex digits of the internal keycode are (N-1) M for an unshifted key and (N-1) (M+8) for a shifted key. The decimal equivalent of the resulting byte is then 16* (N-1) + M or 16* (N-1) + (M +8). The bit number (where 0 means the leftmost bit, as with flags in the d-register) is 36-8* (N-1) - M whether or not the key is shifted. There is a slight anomaly, however, in that for keys with user keycode ± 42, ± 43, or ± 44, N-1 must be replaced by N in the above expressions. This is accounted for in lines 64-72 after which we have in X the absolute value of the user keycode decreased by one only if the original absolute value differed from 44 by more than 2. Thus, X contains the digits M (N-1) or MN, whichever is appropriate--call it MN'.

After lines 73-81 have been executed we have T=M.N', Z=Y=8*N', and X=L=8. In lines 82-83, Y is changed to 8*N' + 8 for shifted keys. In lines 84-88 we isolate M and check for nonexistent rows, branching to an error routine if M is zero or greater than 8. After lines 89-92 we have Z=16*N' +M (16*N'+M+8 for a shifted key), Y=8*N'+M, and X=8. The decimal equivalent of the desired internal code is therefore in Z, and subtracting the number in Y from 36 will give the required bit number (0 to 35). However, the bit is going to be set by stuffing the ⊦ - or e-register into the flag register (d) and setting the appropriate flag, and the calculator does not allow us to set flags beyond 29. This problem is dealt with in lines 93-97 by adding 8 to Y (which will amount to subtracting 8 from the bit number), and also setting flag 08, if Y is less than 8 (corresponding to a bit number greater than 28). Flag 08 will later signal the program to shift the bits to the left by 8 bits before putting them into the flag register, thus bringing bits 29-35 into the range where the flags can be manipulated.

(Some steps could be saved by appealing to routine IF , but the resulting program would be noticeably slower.) In lines 98-101 we finish calculating the bit number--actually the negative of it--and branch to an error message if the bit number would be negative, corresponding to a nonexistent column in the original keyboard.

In lines 102-112 we recall the appropriate status register of bits, shift it one byte to the left if flag 08 is set, and put it into the d-register. If the appropriate bit was already set, then a branch is made to an error message (lines 113-114) to indicate that the key already has something assigned to it; if you want to deliberately make multiple assignments to the same key (say, for experimentation purposes), you can copy **MK** into program memory and delete these two lines. If there is no error branching, then the bit is set, the bit string reshifted to its normal position if flag 08 was set and the results stored in RH or RE (lines 115 through 126). (Note that ARCL 10 in line 120 is simply to effect a 6-byte shift, as the contents of R_{10}, originally created by **OM** are always 6-character alpha data.)

In lines 127-148 we actually construct and store the key assignment in the assignment registers. The procedure depends on whether flag 10 is clear (in which case we put the new assignment in the "left half-register" with three null bytes in the right half, save the assignment in R_{11} for use in future assignments and set flag 10) or set (in which case we get the saved assignment from R_{11}, append the new one to it, store both in the proper assignment register, and clear flag 10). Line 127 puts the bytes 2A 2A F0 into alpha. If flag 10 was set, then (lines 128-129) R_{08} is appended, (shoving the already-existing 3 bytes to where they will not be used); R_{08} contains the 6-character alpha data with bytes 2A 2A F0 a_1 a_2 a_3 where a_1 a_2 a_3 is the old assignment with which the new assignment is to share the register. The new assignment bytes are appended in lines 130-135, and in lines 136-141 flag 10 is reversed with the proper addition of nulls and storage in R_{11}, performed if flag 10 was clear. If $b_1b_2b_3$ represents the bytes of the new assignment, then the contents of alpha line 141 are (where / denotes the boundary of the N and M registers):

2A 2A / F0 b_1 b_2 b_3 00 00 00 if flag 10 was originally clear,

ASTO 'd in R_{11}

2A 2A F0 2A 2A / F0 a_1 a_2 a_3 b_1 b_2 b_3 if flag 10 was originally set.

In either case the M-register ([-register) contains the proper bytes to be stored in the assignment register, and this is done in lines 142-148 by temporarily lowering the curtain to absolute address 16. The assignment is now complete; it only remains to clean up the stack and alpha (lines 149-150), and increment the assignment-storage index in R_{09} if flag 10 is clear (indicating that the next assignment will be in a new assignment register). If R_{09} is incremented and the .END. is reached, then line 153 (SF 20) will be skipped and flag 20 will remain clear; otherwise flag 20 is set. At this point a RTN is executed in the case of the non-prompting versions (lines 154-155); otherwise we get onto lines 156-157 which send us back to LBL 03 (line 32) to prompt for a new assignment, unless flag 20 was not set in line 153, in which case the "DONE, NO MORE" message appears to warn the user that the assignment he just made was his last (lines 158-160 and 165-171). Flag 09 is also set here, so as to obtain a number-of-free-registers display if R/S is pushed and a new register check indicates that there is room. After either the "DONE, NO MORE" or "NO ROOM" message, if the user makes adjustments which do not move the program pointer (such as deleting assignments or reducing the size--not below 12), he may simply push R/S which will lead (via line 172) to a new register check and a continuation of the program.

In the case of the "NO SUCH KEY" error display (LBL 08, line 173) or the "KEY TAKEN" display (LBL 09, line 176), the user is prompted for a new keycode, with the old keycode sitting in X for convenient inspection (lines 187-192). Pushing R/S will enter X as a new keycode and go back to LBL 01 for a new register check (lines 193-194); flags 09 and 20 were already cleared in lines 180-181 to make sure that the prompt for a new input will be skipped and that a register check will be carried out if, say, **+K** is executed. Thus, in "KEY TAKEN" case the user may simply delete the old assignment to that key and push R/S.

In the case of the non-prompting (programmable) versions, all error message displays and halts will be suppressed and replaced by a simple RTN if flag 25 was set beforehand by the user (lines 166-167 and 182-183). If an error then occurs, flag 25 will be cleared (similarly to with the machine functions) and the error message will be in alpha, so that the user can tell whether there was an error and what kind it was and have his own program act accordingly.

CONTRIBUTORS HISTORY FOR MK

THE EARLY DEVELOPMENT OF THE PPC KEY ASSIGNMENT PROGRAMS

by J.E. McGechie (3324)

Ka, *more correctly KOI, ancient Egyptian conception designating part of a human being or of a god. There is still controversy about* ka, *chiefly for lack of an Egyptian definition; the usual translation "double" is incorrect. Written by a hieroglyph of uplifted arms, it seemed originally to have designated the protecting divine spirit of a person, and later the personified sum of physical and intellectual qualities constituting an "individuality". The* ka *survived the death of the body and could reside in a picture or statue of a person.*

Concepts of soul. First of all there is the ka, *which, created with a person, can be defined as "vital force" and outlasts man...*

(The New Encyclopedia Britannica, V 649 & 6,506.)

The beginnings of synthetic key assignment techniques properly belong with the PPC Club and with Richard Nelson, its founder, guide and leader. To say this is not just to pay a generalized kind of tribute, though that would not be at all inappropriate, for it is his conception of the role of a member that has guided the PPC Club's development, the most important aspect of which has been the supportive role its members may play in relation one to another: may play, and thanks to him, do.

First, a little backtracking, a little history. The HP-41C was released in July, 1979. It was announced and described in the July issue of the *PPC JOURNAL* in proper terms of gratitude and pleasure at the excellence of its conception and design. PPC members being what they are, a curious bunch (pun intended), the bugs of the early machines were found almost immediately, B2, the 'wrap-around', indirect storing into program memory, being the most important. This was found first by Bill Danby (1213) of the Chicago Chapter-the Chip Chapter-and passed on to the waiting PPC world through the *PPC JOURNAL* in August. Many of its novel properties were sorted out by Jim Horn (1402) and spread around to the faithful at the 1979 PPC Conference. From Jim we learned how to recall 'NNN's' from program memory, using B2, and armed with the first byte charts, the Corvallis prefix and postfix tables appearing in the Corvallis Column, we were able to selectively recall program bytes as text string, as forbidden alpha characters in the stack registers.

The 41C was a challenge. Here we had the equivalent of the Black Boxes of HP-67/97 fame, but without the tedious hassle which that device required, or the surgery needed for the phase-interrupt Block 3 access. In addition, the gods of Corvallis had handed the tablets of clay, holding the hex codes, to their faithful disciple, Richard, and Richard had given them to us. No stumbling down the mountain for <u>him</u>! Then John Kennedy wedded the charts, producing the unified hex table, essential for all the spells that were to be cast for the next year or more.

I had become hypnotised by NNN's in the previous year, found the sign digits of the HP-19C, wrote voluminously to Joseph Horn (1537) and George Istok (2525) about them, their properties, the clues that NNN behaviour provided to the internal workings of HP machines. Both patiently taught me, through their letters, protesting at the bulk of mine, what they knew and how they had found it out, and I read and reread everything in the *65 NOTES* and the 'Journal' touching on the subject, compiling a complete bibliography, only the first part of which has appeared in print.

Many of the HP-41C NNN properties were very like those of the earlier machines (and again like those of the earlier machines, <u>still</u> lack proper documentation!), but the program code/NNN and normal number correspondences were crystal clear; thanks to Corvallis and the grizzled early NNN pioneers from the first months of the 65 Users onwards to the last months beforty-one C.

I was fortunate enough to get my hands on an early model, complete with those lovely bugs, and within two days I had stamped out the .END. and found to my amazement that a just master-cleared 41C had instructions in it! At first I thought I was in ROM, but these could be deleted and replaced, sometimes with MEMORY LOST, and I stepped through them, listed them by hand, sent long speculative letters to Richard Nelson, John Kennedy, Joseph Horn, George Istok, who patiently replied with other mysteries. Then I noticed some deletions and insertions made the flag annunciators go on and off, and realised what was happening: I was reading, amongst other things, the flags/bits of the flag register as program, and rang Richard in California from Melbourne. Good to speak to The Master Himself, but he seemed (no surprise) to know all about what I had found: "Oh, yes, we know all about that, there's an article coming out in the next 'Journal', by Bill Wickes (3735) describing those registers..." Around $18 later, having hung up, I found the alpha register in the same way, and then the stack, all in <u>program</u> mode. Richard had been talking about the first synthetic instructions' access to status, and thought I was also. I had been talking about program access to status, and thought Richard was too, though I put my puzzlement over some of our cross purposes to ignorance or obtuseness on my part.

Several days later the promised 'Journal' arrived with Bill's revolutionary first article in it. I realised what had happened in the Melbourne/Sant Ana conversation, and set about climbing up on Bill's shoulders to look at the new prospects, worked through the Corvallis columns again, rang Graeme Dennes (1757) for another two hour conversation...

Bill had made the first synthetics by B2 storing of normal numbers into program space, and those which couldn't so be keyed in, by fancy reading in of HP-67 fabricated cards. But by setting and clearing the flags of the flag register, as I had found, any byte could be made and seen as instruction or instruction component while reading status as program. <u>So could the crucial RCL d</u>. Thus one day, again while on the phone to Graeme, I keyed into the flags a RCL d, single stepped on it, to recall its code from the flags as part of an NNN, and used B2 to place it in regular program memory. Now it was easy to make any of the needed synthetics in the same way, without needing to be in status in program mode. Between us, Graeme and I soon had a program tinkered up to manufacuture, under program control, any sequence of bytes from the hex table, and store them in program memory in a sequence of registers using B2. The results could then be studied at leisure to see what they might do.

I rang Richard once more, to describe this bootstrapping method briefly, but international calls at $15 or more for three minutes leave little room for maneuvering, and was urged to look for ways of assigning the synthetics to keys. The early way I had used for synthetics needing the digits from A to F had been by B2 recall of suitable nonsynthetics from program memory as alpha strings, and to bootstrap, still needed this. (I had no printer, memory modules, no card reader--all were still very rare in those days.) Then these individual bytes, as alpha characters, could be separated in alpha and stored for future assembly.

All of this work, with details of the operation and attempts at the explanation of what was happening, was written into one complete, but 20 A4 paged article sent to Richard early in December 1979, and in the opening parts of two others, also sent to him. All in them were very soon to become obsolete, and some were made so by Bill Wickes' (3735) RAM safari article and Black Box program set in the December 'Journal'. And remember, too, that he was the first to make synthetic assignments by deleting assignment bytes in the assignment registers, and replacing them, AND, more importantly, describing with clarity, to the rest of us how, why, and what he had done. <u>That</u> was the essence of our PPC.

The second call to Richard was early in December, and the third was made on the 29th. On that day I sat down at the typewriter to describe how to assign synthetics to keys manually. Cath, my wife, wondered why I was looking so smug, and when I told her, encouraged another California call. I gathered that others had already done the same... The two pages of that detailed typed letter to Richard turned into 6, when I started to think of programming methods, and the first KA ran, debugged though needing Bugs 2 and 3, about 24 hours later. The letter, still typed, grew to 12 pages and was mailed to Richard on January 2nd. I thought it likely he would publish extracts from that letter, mostly written in the form of an article, and sent copies to George Istok, John Kennedy, Joseph Horn,...with cards for the first KA together with newly made status cards, using a reader lent by local HP.

I'm not sure whether that letter ever reached Richard, and was disappointed when none of it appeared in the 'Journal' in January, despite Richard's mention of the early KA in the yellow newsletter. It explained some of the immediate background better than I can now--here are a few extracts:

"Dear Richard--or should I say--Dear Master,

"It is accomplished! I have just been excitedly keying in dozens of RCL d and X<>M directly into program! The first is assigned to 1/x, and the other ΣC. Since I spoke to you last week or so, I have been thinking about the problem you sent me then--to work out a way of assigning the NN functions to keys, so that they could be keyed directly into program in USER mode. I had read the latest article by Bill Wickes many times--it is not free of mistakes, completely understandable in such a difficult area, but it provided half of the essential clues. The rest of them, as you should by now have guessed, came from a letter from George Istok, dated 5th December, but not posted until December 13th. I don't have a card reader, but he had enclosed a card on which he said he had assigned TONEs to the number keys. I quote from that letter:

"'Those tones are assigned directly to the keys. XROM 60? It is not the XROM code from the HEX table. Clear the .END. and step through what you find. Typically, you will see ⊤, TONE #, XX, TONE #, XX.

XX represents a single byte instruction. That single byte instruction is the address of the key to which the tone is assigned. Do not change any address yet. You can clear a TONE # and insert most two byte instructions, but be very careful or full is what you will be and then over full and then MEMORY LOST. The ⊤ is HEX F0 and seems to be used as a marker. Hex F1 works as well and I think only the F is necessary for the micro-processor. If you want to change the function assigned to key 0, to say -, insert LBL 03 after deleting TONE 0 and then insert -. LBL 03 is the normal blind for this byte."

"George then went on to give a list much like that given by Bill Wickes on *PPC JOURNAL*, V6N8P28d, but he does not interpret them all in quite the same way. He goes on to note that the 9064 and 9065, 9164 and 9165, displaying as RCL 00 and RCL 01, STO 00 and STO 01, are in fact direct functions on registers 100 and 101, as Graeme Dennes and I had discovered, and mentions other interesting things, but <u>THE</u> important things related to the key assignment registers.

"Without a card reader I had to puzzle out what he meant from his letter alone. Then I assigned some functions, deleted the "permanent" .END. as he had suggested and looked at the program lines read from those registers. The registers are of course assigned in the kind of way that one would expect."

I then read and reread Bill Wickes notes on key assignments, read George's letter several more times, tried keying in two byte functions into the right places in the assignment registers as George, I figured must have done--then--MEMORY LOST, and painful bootstrapping again using B2. Having described this, I went on:

"Having got going again (my + key is standing up under the strain remarkably well), I decided to take a leaf out of Bill Wickes' Black Box and build up the necessary NNN in the alpha register by creating the necessary bytes one by one, storing them as alpha data in the user registers, assembling the seven bytes in the register M section of the alpha register, using a RCL M to transfer them to the X register whence they could be stored straight into register 192 (0C0) by an indirect store using B2.

"So I did, and to my enormous delight, it worked. Here are the details--from scratch."

No point now in giving those details--briefly, they were recalled from program memory by B2, isolated in the alpha register, one by one, stored, eventually alpha recalled to M... It took three hours of hex chart consultation and muddle. The method, though crude, though it <u>should</u> have used Bill's "Build" was general, and repeatable. After typing two pages, I rang Richard to tell him it could be done, then back to the typewriter to describe the method. After six pages, the current heat wave got at me, and I slept off the binge, but just before I did:

"The next thing to think about is whether the key assignments can be generated by a suitable program, just as the two or more byte instructions can. (Routines for doing so are given in the *BUILDING BYTES FROM BITS* article.) I won't delay this letter to think about and try out ideas, however, or it might suffer the fate of the other article material I have been working at for the last five or six weeks, having started with the idea of getting complete, if hastily written articles off quickly before someone else worked out the same ideas. It now seems to me that it is necessary to get up <u>very</u> early in the morning--or probably not to go to <u>bed</u> at all--and that has happened too--to beat our membership. But at least Graeme and I have been almost keeping up with the rest of you over there."

I woke the next morning with some ideas (freebie work--hand it over to that which thinks in you by working like hell, then sleeping it off. Often the problem is solved, or nearly so, with no further conscious effort). 12 hours later I had KA#1 working. It allowed the assigning of 14 keys in about three minutes, and slightly modified, 40 could be assigned on the basic machine. Here is that first program. The "digits" called for were the successive six digits for the assigned function and the code of the assigned key. 10 to 15 were keyed in for hex A to F.

HP-41C NON-STANDARD KEY ASSIGNMENT PROGRAM "KEY ASSIGNMENTS"

This program requires a machine with both B2 and B3, and assumes a SIZE of 017 on the basic machine with NO periphals plugged in. Since bad crashes are possible, it should not be used without complete understanding of the key-assignment register codes.

I described the use of the program, precautions necessary, and then tried to explain how the 41C found and executed, or placed key assignments in program. (Discovery is often useless without attempted explanation, and further discovery sometimes inhibited.) Some was right, but some were blushworthy howlers... All this was then quite new and strange territory. I like explanatory diagrams, and included several. Here is one, used to describe program operation:

> The two byte function code and the key identification code are generated in the flags in a single run of sub-routine 3B ('Three Bytes'), but in the right format to be recalled as alpha data; the first byte is set to hex 10 by setting only flag 03. These are the bits (flags), digits and bytes involved:

	13				12				11				10			
Bit (flag) no.	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Bits (flags) set	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0
	1				0				9				0			

The alpha data prefix byte — 'DIGIT 1' 'DIGIT 2' 'BYTE 1' (Prefix)

Reg. digit no.	9				8			
Bit (flag) no.	16	17	18	19	20	21	22	23
Bits (flags) set (1) or clear (0).	0	1	1	1	1	1	1	0
	7				E			

'DIGIT 3' 'DIGIT 4'

'BYTE 2' (Suffix)

Reg. digit no.	7				6					
Bit (flag) no.	24	25	26	27	28	29	30	31	32	33
Bits (flags) set (1) or clear (0).	0	0	0	1	0	0	0	1		
	1				1					

'DIGIT 5' 'DIGIT 6'

'BYTE 3' (Key code)

> The method used is that of the two byte instruction generating routines in my Building Bytes From Bits. If the 'digit' keyed in is odd, the least significant bit of the digit is set (the highest flag number of the digit). If the integral part of half of the digit is odd, the next most significant bit is set--and so on. (The same method, but in reverse, I was pleased to find, is used by Bill Wickes in his UNBLD, and it is in effect used in his REBLD. See *PPC CALCULATOR JOURNAL,* V6N8P30.)

The routine used actually dated from even earlier methods developed for writing programs for the HP-19C, teaching it to work problems in elementary logic, used there to generate a truth table scan for not only binary logic, but also for many-valued logics. So? Programmers' paths are very devious!! I ended this long letter by describing how to key in the program using only B2, and described how some of the synthetic assignments behaved, and the correspondence between the assignments and the XROM numbers, tried more explanations , and made more errors, noticing the XEQ, GTO and W Q-loaders, and that their alphas came from previously alpha keyed instructions, something we were not to begin to understand for about six months. I thought then that we had 256 x 256 assignments to explore, and started a few weeks later with an enormous sheet of paper, crawling over it with the 41C in hand. But I started at the top left hand corner... So Bill discovered the byte jumpers, and Valentin Albillo (4747) the alpha label Q-loader, and I missed the local alphas then (e.g. LBL X, XEQ P, etc.), though not later. It was the start of one of the big explorations, and my map of ½" by 1" rectangles never grew beyond about 6' x 8'. The 256 x 256 shrank quickly--to 16 x 16 again.

Just after all this excitement, when several generations of KA had come and gone, Ron Eades (4417) lent me his brand new reader and printer for three weeks, and cards flew with Quantas and Pan Am to friends, but I wrote letters, not articles. Our local HP lent a reader for a few days early in January, and the first KA went out to friends on cards.

Though I heard nothing from Richard, John Kennedy ran the program, and after some initial troubles, was soon assigning the whole 41C keyboard with its aid. He wrote further routines for it, which would allow the byte numbers to be keyed into the stack, including that for the keycode, and without sending the revisions, described them and argued for their convenience. He was right--it made things much easier. This first program was replaced by KA #2, then by KA #3...

Graeme Dennes had B2, but not B3. In the first weeks of January the three bytes for the assigned function and its key code were made in the flags in a form allowing them to be recalled as an alpha string of three characters, but that needed B3. (The byte numbers themselves were decomposed into binary by repeated division in a loop, the current bit/flag being set when the remainder was non-zero, a decimal to hex routine similar to that used in the first Black Box set of programs by Bill Wickes.) The flag numbers had to run up to 35 when keys on the left hand side of the board were to be assigned.

As can be seen from the program above, KA #1, entry was unecessarily fiddlesome, and required constant reference to a detailed keyboard map I sent with that original letter. Errors were too easy, and Graeme and I bent our minds to using the Wickes found and documented algorithm, not quite correctly given in his Jungle Report in the December 'Journal', which led from the key codes to the digits of the key coding byte of the assignment register and of the global label. John Kennedy had sent us a map of the flag numbers set in e or ⊦ when a key was assigned, drawn on the large keyboard diagram contributed by Bill Kolb (265). I don't know where these had come from, but they were absolutely essential to determining the algorithm, complementing that from the key codes to the assignment bytes, which led from those codes to the flag numbers.

It was mid-January. Then, under strict orders from the household boss, and pressure from the children, the 41C was left behind in Melbourne while our family went to its annual two weeks at the beach, but pencil and paper were not forbidden... Both algorithms were cracked there (and simultaneously in Melbourne by

Graeme), a complex stack analysis allowed flag and byte code computation simultaniously in the stack, and an artificial B3 was devised. During the day I made sand castles with the children, at night, stack analyses with myself.

Now B2 was the only bug needed. There things stayed, and the January 'Journal' appeared. Every successive revision went to George Istok, John Kennedy, Joseph Horn, and I wrote to Bill Wickes for the first time, asking which assignment layout of the synthetics he favoured, and which synthetics. Both he and George, and as I later found, Bob Edelen (339) and Bill Kolb (265), had also found manual ways of (B2) assigning keys. Bill sent me his artificial B2 method which went into KA #9, and two revisions later the experimental program KA #11 was published in the 'PPCCJ'. It was almost farewell to key assignments for me then, but when Tom Cadwallader's (3502) revision of KA #11 appeared, I combined its best features with those of KA #13 and sent it to him. Our correspondence has continued ever since. That last improvement reduced initialisation from 20 to 2 seconds, by using synthetic text lines for the artificial register c contents construction, and for the necessary assignment register prefix byte, F0.

Richard Collett (4523) here, Geoff Smith (5307), and others overseas took over. I stayed with my KA #14, which could also make any NNN by feeding it with byte numbers, and used the BLDSPEC function for fast byte making. Until the ROM sample arrived here, it served me well. (as always, that synthetic alpha string method--that, too was Bill's invention. Almost nothing of my own!)

Richard did a herculean job with the KA #11 passed on to him from John Kennedy, but either did not get my documentation, in note form in the letter with which that version was sent (actually only an experimental version, not one intended for use), or possibly misplaced it. When I saw it in print, I had a new revised and shorter, faster version, using text lines, one easily able to be entered. It came with a B2 synthetic line generator, which pulled bytes from one program file, formatted them, and dumped them on top of +'s in the next lower program file, after which single stepping allowed the keying in of the non-synthetic instructions around them. That had six pages of article written (typed, even!) for it when Tom Cadwallader's version appeared in the next issue--and was scrapped... This was a recurrent problem: no sooner was one ready, than a revision improved it. The notes Richard properly promised with KA #11 had to be revised as fast as they were written...

From this point on, others know more of KA than I. Richard Collett (4523) wrote more elaborate, and faster, more convenient routines than mine, generated a counter for the assignment pairs to stop assignments before the .END. was over run, organized a method of recalling from 0C0 upwards, checking for zero contents, re-prefixing an F0 byte if not, recalling the next register. Roger Hill (4940) invented the Pack Assignments feature (why didn't Corvallis think of that? No microcode room left?), and others improved his versions. One of Richard Collett's best programs packed the assignments swiftly while looking for the first free register, and told how many there were, and like Roger Hill's, allowed the assigning of one key at a time. Bill Wickes wrote a remarkable version using no user registers at all, all being done in status. Now all will have the hassle free version of the ROM, but if all do not have a ROM, they may use various compact versions such as Geoff Smith's, under 200 bytes in length. It needs manual assigning of two keys at a time, whose assignments are then overwritten, then two more...

Finally--a disclaimer. Here are the people who really wrote the first programs: George Istok, Bill Wickes (major one), Richard Nelson, who probably realized before any of us that any two-byte assignment was possible, and had to be so if any XROM, eXternal ROM functions were able to be assigned, Joseph Horn, who knew the most, with his brother, Jim Horn, about NNN's, John Kennedy for stack entry, Graeme Dennes for always insisting on improvements, and usually then making them himself, and Bill Wickes again and again and again. He didn't approve, with George Istok, of program-assigned synthetics. It was simple--using his wits, and his Black Boxes, to assign synthetics. George thought those not familiar with the full operation of NNN's etc., on the 41C were advised to keep away from synthetics, and from the trouble they could occasion the careless user. I think and thought George was wrong, and fought battles by mail with him. The ignorant should be educated at the same time as synthetics use is made simple and obvious for them, just as with any advanced programming technique.

Of course, before any of these people, there were the HP-65 pioneers, the HP-67/97 investigators, the HP-45 bed-bouncers (to start that clock), the key pokers and pressers, and the curious who noted the quirks and looked for their uses--all tied in the communicating network of the master crouched in Santa Ana, manipulating all we puppets for our own, ultimate, good.

Postscript on MK History

The PPC ROM obviously had to have a state-of-the-art key assignment program. Three superb versions were written, each with its own set of advantages. Tom Cadwallader (3502) wrote a version (*PPC CALCULATOR JOURNAL*, V7N10P19) which totally eliminated the need for data registers. This version was subsequently updated by John McGechie (3324) and Richard Collett (4523). Richard Collett also wrote a SIZE 001 version which had extensive error checking and ran very quickly. Since this version has not been published yet, it is listed here as a learning tool for ambitious synthetic programmers. Roger Hill (4940) wrote the ROM version of MK. It uses six data registers and occupies more bytes than either of the other two versions, but it has two offsetting advantages. It is virtually "bulletproof" against user errors and it has a subroutine capability. Richard Nelson (1) felt very strongly about going all-out for the most "friendly" and flexible MK. It was a great source of comfort in the ROM development to know that if we really needed 200 more bytes, we had two alternate versions of MK. The alternate versions by Tom Cadwallader and Richard Collett are masterpieces in their own right and worthy of study.

RICHARD COLLETT'S KA#18

As most PPC members will be well aware, the history of the key assignment programs sums up the dedicated, enthusiastic international cooperation of the Club, always led and powered by our leader, Richard Nelson. The first key assignment program, written by John McGechie (3324) in the last days of 1979 was #1. Many contributed to its evolution up to the first published versions in the February and March Journals in 1980. John McGechie stopped writing improved versions with his KA#14, but Richard Collett (4523) continued the series, sending his KA#16 to Keith Jarett (4360) with a Melbourne revision of the December version intended for the PPC ROM, that version having been adapted and streamlined for the ROM by Roger Hill (4940) and many others. Richard Collett continued to refine the program variety he wrote then, and the most polished version, though like all programs, open to further streamlining, was the one which appears below, KA#18. Until the ROM was out, this has been in regular use by Melbourne Chapter members, who agree that it is by far the best of all, more convenient in some ways than that in the sample ROM's they have been privileged to have and to use for trial purposes.

Here is Richard Collett's description of KA#18:

Executing "KA" first causes the assignment registers to be packed. The routine PA which effects this, returns information identifying the highest numbered assignment register used, when flag 09 set indicates that this register contains only one assignment, that it is only half full.

Having completed this packing, and displayed how many assignment registers are used, the display now prompts for the PREFIX-POSTFIX-KEYCODE with the display "KEY nn", where nn indicates which key assignment this will be. If seven keys have been assigned, that for which the user is prompted will be the eighth.

Entry of this assignment information by the user is in the usual stack format--as prefix decimal byte number, ENTER, postfix decimal byte number, ENTER, keycode, R/S. There is one limitation here: the program will not handle a zero prefix, and consequently lines 35 and 36 turn such an entry into a 1.

As the bytes are made by the program, their numbers and the subsequent keycode are displayed to the user, forming a useful check on correct entry.

The keycode is first converted to the corresponding flag number, tested for legality, the corresponding flag (in e or ⊦) is tested to see whether it is already set, and if not, set. The flag number is then converted to the keycode byte required in the assignment register. During the flag testing and setting process, the alpha string being built is held in the stack, consisting at this stage of the prefix and postfix bytes.

The program requires one user register only, holding the index for the top assignment to date, together with the .END. address.

Some unique features of the program.

Unlike other key assignment programs, the previous half, if any, of the assignment register contents does not have to be held in storage. This is due to the method of adding the new assignment effected in lines 125 through 142 of the program.

Since the registers are always packed during the initialization process, any cleared registers are recovered before new assignments are made.

The codes being processed are displayed to the user during processing.

The number of assignments are displayed to the user by the prompt for the next, and the number of registers used are displayed during the initialization. These are desirable when assignment cards are being prepared, the maximum per card being 32.

The whole program will fit on 6 tracks, and is only 670 bytes. Though there are shorter key assignment programs, they lack many convenience features, and can be risky to use, there being dangers of overwriting existing assignments, leading to default functions, or worse--to overwriting the .END. and the contents of the last program file. However, this is gained at the cost of requiring at least one single density RAM module.

The program retains the useful feature of John McGechie's earliest programs, an NNN maker fed a sequence of decimal byte numbers up to seven, to make any NNN by that means. Some prefer this as a manual NNN maker, to the ROM version, pioneered by Bill Wickes (3735), requiring 14 digits to be specified as their alpha counterparts in the alpha register. This is accessed by executing "NN".

All assignments may be quickly cleared by execution CA . On completion, there is a return to the entry for new assignments. As before, this is convenient when making assignment cards for special purposes.

For assignment register access, the curtain is set at decimal address 001, i.e., R00 is effectively register Z of the stack. This allows direct storing into any of the status registers, apart from T, itself readily accessible in the normal way. This is used at lines 234 and 235. (It must be remembered that recall of status under these circumstances, recall as if from user registers, will cause normalization. Only recall of status by direct register address circumvents this.)

User Constraints: There must be an END above "KA" in Catalog 1 to allow reverse branching with the curtain lowered. Although "KA" is interruptible, don't switch into PRGM mode or SST. The calculator will not be able to compute a correct line number, lacking the Catalog 1 label linkage. As a result, the program pointer will be changed and you'll be out of business.

As it is presented here "KA" is not compatible with MK. Unlike "KA", MK allows the first byte of an assignment to be zero (null). This "missing" byte disrupts the packing operation of "KA". If you plan to use "KA" either start with CK or CA or make sure that you didn't use MK to make any assignments with a null first byte.

Those wishing to study the structure of this program will find much that is unusual and ingenious in it, but they should be aware as they do of the many keypunching fingers whose activity led to what is here. They hang from the hands of John McGechie (3324), John Kennedy (918), George Istok (2525), Richard Nelson (1), Roger Hill (4940), Tom Cadwallader (3502), Bill Wickes (3735), and Keith Jarett (4360), and from the hands of many others who refined the component routines built into this program.

Program Analysis:

Lines 01 thru 06	Pack keys (LBL 04) to find top assignment register address (stored in R00).
Lines 07 thru 10	Is a new assignment register required?
Lines 11 thru 15	Error message when no assignment space left. GTO KA effects packing to search for space.
Lines 16 thru 28	Calculate number of the current assignment, and
Lines 29 thru 32	prompt for input.
Lines 33 thru 42	Compute prefix and postfix bytes, testing for zero postfix at lines 35 and 36.
Lines 43 thru 67	Compute flag number from keycode, correcting for keys 12 to 14 (-12 to -14).
Lines 69 thru 70	Test whether legal key code.
Lines 71 thru 76	Test whether system flag.
Lines 77 thru 100	Set flag in ⊦ or e, testing whether key already assigned (exit to error message at line 90).
Lines 106 thru 123	Compute assignment register keycode byte number, generated by the decimal-to-character routine at lines 156 to 183.
Lines 125 thru 142	Load assignment bytes in the highest free register.
Lines 143 thru 152	Error messages.
Lines 153 thru 184	NNN maker and decimal-to-character routine.
Lines 185 thru 192	Common Initialization.
Lines 193 thru 224	.END. address finder. Line 225 gives the curtain lowering contents for Register c.
Lines 227 thru 245	Clear assignments.
Lines 246 thru 255	Initialize for packing assignments.
Lines 259 thru 264	Recall assignment register contents.
Lines 268 thru 270	Is register empty?
Lines 271 thru 275	Is the left hand key code zero?
Lines 276 thru 281	Is the right hand key code zero?
Lines 282 thru 284	Is either keycode zero?
Lines 285 thru 288	Rebuild assignment register contents.
Lines 289 thru 295	Restore assignment register contents.
Lines 296 thru 300	Increment storing counter, ISG T, and recalling counter, ISG Z.
Lines 301 thru 305	Is there a half register free?
Lines 306 thru 316	Join two assignment register assignment contents.
Lines 319 thru 322	Are both keycode bytes clear?
Lines 323 thru 328	Abstract non-cleared assignment bytes.
Lines 331 thru 334	Store half assignment register contents.
Lines 338 thru 344	Calculate # of registers occupied by assignments.

Most of the synthetic lines are familiar here. Those that do not reveal their contents in the printout are as follows:

Line 82 is F1 01, line 127 is F5 01 69 00 10 00, as is line 225, line 129 is F1 F0, as is line 286, line 130 appends three nulls, line 195 is F6 7F 00 00 00 00 02, and line 308 is also F1 F0. Line 151 is 9F 2A (TONE 42).

BAR CODE ON PAGE 485

```
01♦LBL "KA"
02 XEQ 04
03 RDN
04 STO 00

05♦LBL 25
06 DSE 00

07♦LBL 21
08 FC? 09
09 ISG 00
10 GTO 22

11♦LBL 05
12 "FULL"
13 BEEP
14 PROMPT
15 GTO "KA"

16♦LBL 22
17 RCL 00
18 191
19 -
20 INT
21 ST+ X
22 2
23 FC? 09
24 SIGN
25 +
26 "KEY"
27 FIX 0
28 ARCL X
29 FIX 4
30 TONE 3
31 4
32 PROMPT
33 FIX 0
34 X<> Z
35 X=0?
36 SIGN
37 CLA
38 VIEW X
39 XEQ 08
40 RDN
41 VIEW X
42 XEQ 08
43 X<>Y
44 VIEW X
45 X<0?
46 SF 19
47 ABS
48 41
49 X<Y?
50 X=Y?
51 DSE Y
52  E1
53 +
54 X<=Y?
55 DSE Y
56 RDN
57 LASTX
58 /
59 INT
60 LASTX
61 FRC
62 80
63 *
64 +
65 CHS
66 36
67 +
68 ENTER↑
69 X<0?
70 GTO 06
71 24
72 X<=Y?
73 SF 20
74 X>Y?
75 CLX
76 -
77 RCL [
78 FS? 19
79 RCL e
80 FC? 19
81 RCL '
82 "x"
83 X<> [
84 STO \
85 FS?C 20
86 "⊦***"
87 X<> [
88 X<> d
89 FS? IND Z
90 GTO 07
91 SF IND Z
92 X<> d
93 X<> [
94 ASTO [
95 ARCL ]
96 X<> [
97 FS? 19
98 STO e
99 FC? 19
100 STO '
101 X<>Y
102 CLA
103 STO [
104 RDN
105 CLX
106 8
107 ENTER↑
108 72
109 R↑
110 4
111 +
112 ENTER↑
113 R↑
114 MOD
115 LASTX
116 RDN
117 -
118 ST+ X
119 LASTX
120 +
121 -
122 FS?C 19
123 +
124 XEQ 08
125 RCL 00
126 ASTO Y
127 "xi♦8♦"
128 RCL [
129 ""
130 FC? 09
131 "⊦♦♦♦"
132 ARCL Z
133 X<> c
134 FS? 09
135 ARCL IND Y
136 RCL [
137 STO IND Z
138 RDN
139 STO c
140 FC?C 09
141 SF 09
142 GTO 21
143♦LBL 06
144 "INVALID"
```

```
145 GTO 00
146•LBL 07
147 X<> d
148 "USED"
149•LBL 00
150 AVIEW
151 TONE 2
152 GTO 22
153•LBL "MN"
154 CLA
155 PROMPT
156•LBL 08
157 INT
158 256
159 MOD
160 LASTX
161 +
162 OCT
163 X<> d
164 FS?C 11
165 SF 12
166 FS?C 10
167 SF 11
168 FS?C 09
169 SF 10
170 FS?C 07
171 SF 09
172 FS?C 06
173 SF 08
174 X<> d
175 ASTO [
176 X<> [
177 "├*"
178 STO \
179 "├*"
180 X<> \
181 CLA
182 STO [
183 RTN
184 GTO 08

185•LBL 09
186 CF 09
187 CF 10
188 CF 19
189 CF 20
190 CF 21
191 SF 29
192 AVIEW
193 RCL c
194 STO [
195 "├****x̄"
196 RCL [
197 X<> d
198 CF 00
199 CF 01
200 CF 02
201 CF 03
202 FS?C 07
203 SF 05
204 FS?C 08
205 SF 06
206 FS?C 09
207 SF 07
208 FS?C 10
209 SF 09
210 FS?C 11
211 SF 10
212 FS?C 12
213 SF 11
214 X<> d
215 DEC
216 2
```

```
217 -
218 191
219 X<>Y
220 X<Y?
221 GTO 05
222  E3
223 /
224 +
225 "*i*0*"
226 RTN

227•LBL "CA"
228 "CLEAR A"
229 XEQ 09
230 STO 00
231 RCL [
232 X<> c
233 .
234 STO 09
235 STO 14
236 RCL b
237 X<>Y
238 STO IND T
239 X<>Y
240 ISG T
241 STO b
242 RDN
243 RDN
244 X<> c
245 GTO 25

246•LBL 04
247•LBL "PA"
248 "PACK A"
249 XEQ 09
250 ENTER↑
251 FIX 8
252  E
253 -
254 .
255 RCL [
256 GTO 12

257•LBL 10
258 CF 20
259 X<> c
260 X<>Y
261 X<> IND Z
262 X<>Y
263 X<> c
264 X<>Y
265 CLA
266 ARCL [
267 STO [
268 SIGN
269 X≠0?
270 GTO 01
271 "├****"
272 ASHF
273 X<> \
274 X=0?
275 SF 19
276 X<> L
277 STO [
278 ASHF
279 X<> [
280 X=0?
281 SF 20
282 FC? 19
283 FS? 20
284 GTO 02
285 X<> [
286 ""
```

```
287 ARCL X
288 X<> [

289•LBL 11
290 X<>Y
291 X<> c
292 X<>Y
293 X<> IND T
294 X<>Y
295 X<> c
296 ISG T

297•LBL 12
298 ISG Z
299 GTO 10
300 X<>Y

301•LBL 01
302 FC?C 10
303 GTO 03
304 SF 09
305 CLA

306•LBL 13
307 X<> [
308 ""
309 X<> [
310 X<> \
311 ASTO [
312 ASTO [
313 ARCL 00
314 "├****"
315 X<> \
316 GTO 11
```

```
317•LBL 02
318 X<>Y
319 FS?C 19
320 FC? 20
321 FS? 54
322 GTO 12
323 "├*"
324 FC?C 20
325 "├***"
326 X<>Y
327 CLX
328 STO \
329 FS?C 10
330 GTO 13
331 SF 10
332 ASTO 00
333 X<>Y
334 GTO 12

335•LBL 03
336 R↑
337 ENTER↑
338 INT
339 192
340 -
341 .5
342 FC? 09
343 SIGN
344 +
345 TONE 0
346 CLA
347 FIX 1
348 ARCL X
349 "├ REG USED"
350 AVIEW
351 .END.
```

MKA Melbourne

This key assignment program appeared in PPC CJ, V7N10P19. It was written by Tom Cadwallader (3502) and modified slightly by John McGechie (3324) and Richard Collett (4523). When using MKA you need not make an even number of key assignments, but you must press R/S in response to the prompt for the second of each pair of key assignments. No data registers are used, so SIZE 000 is OK. The same user constraints apply as for KA#18 (END above MKA in Catalog 1, don't SST, don't mix assignments with MK).

Note on MKA listing: The following lines are not represented accurately in the printed listing. Their hexadecimal equivalents are given here.

Line	Hex Equivalent
08	F7 F0 00 00 00 00 00 00
144	F1 F0
285	F5 F0 04 01 C9 01
393	F1 F0
404	F1 F0
416	F1 F0

FURTHER ASSISTANCE ON MK

Call Tom Cadwallader (3502) at (406) 727-6869.
Call Roger Hill (4940) at (618) 656-8825.

BAR CODE ON PAGE 486

```
01♦LBL "MKA"
02♦LBL 17
03 SF 03

04♦LBL 04
05 CF 00
06 CF 01
07 XEQ 00
08 "♦♦♦♦♦♦"
09 X<> [
10 X<> IND L

11♦LBL 03
12 ""
13 X<> [
14 "⊢♦"
15 X<> \
16 X=0?
17 GTO 01
18 "⊢------"
19 X<> \
20 ISG Z
21 GTO 02
22 STO IND L
23 RDN
24 STO c
25 FC?C 03
26 GTO 18
27 XEQ "PKA"
28 GTO 04

29♦LBL 02
30 X<> IND Z
31 GTO 03

32♦LBL 01
33 RDN
34 STO c
35 CLA
36 SF 02
37 CF 03

38♦LBL 15
39 CF 22
40 ASTO L

41♦LBL 19
42 "PRE↑POST↑KEY"
43 TONE 9
44 PROMPT
45 CLA
46 ARCL L
47 FC? 22
48 GTO 16
49 X<> Z
50 4
51 RDN
52 X=0?
53 X<> T
54 XEQ "D↑C"
55 XEQ "D↑C"
56 36
57 STO a
58 RDN
59 ENTER↑
60 X<0?
61 SF 00
62 ABS
63 .1
64 *
65 LASTX
66 -
67 INT
68 8
69 X<Y?
70 GTO 01
71 RDN
72 ST- a
73 4
74 X=Y?
75 SF 04
76 RDN
77 LASTX
78 FRC
79 X=0?
80 CF 04
81 .1
82 FC?C 04
83 CLX
84 +
85 80
86 *
87 ST- a
88 2
89 *
90 +
91 8
92 FC? 00
93 CLX
94 +
95 X<> a
96 X<0?
97 GTO 01
98 24
99 X<=Y?
100 SF 01
101 X>Y?
102 CLX
103 -
104 FS? 00
105 RCL e
106 FC? 00
107 RCL '
108 ASTO L
109 STO [
110 FS? 01
111 "⊢♦♦♦"
112 X<> [
113 X<> d
114 FC? IND Y
115 GTO 05
116 X<> d
117 CLA
118 ARCL L
119 "⊢TAKEN"
120 TONE 0
121 GTO 02

122♦LBL 05
123 SF IND Y
124 X<> d
125 STO [
126 "⊢♦♦♦♦"
127 FC?C 01
128 "⊢♦♦♦"
129 RCL \
130 FS? 00
131 STO e
132 FC?C 00
133 STO '
134 CLA
135 ARCL L
136 RCL a
137 XEQ "D↑C"
138 FS?C 02
139 GTO 15
140♦LBL 16
141 FS? 02
142 GTO 19
143 ASTO X
144 ""
145 FC?C 22
146 "⊢♦♦♦"
147 ARCL X
148 RCL [
149 XEQ 11
150 176
151 R↑
152 STO IND Y
153 R↑
154 X<> c
155 CLST
156 "R/S TO CONT-"
157 BEEP
158 PROMPT
159 GTO 17

160♦LBL 01
161 "⊢ERROR"

162♦LBL 02
163 AVIEW
164 TONE 0
165 X<> \
166 CLA
167 X<> [
168 ASTO L
169 GTO 15

170♦LBL 18
171 CLST
172 TONE 0
173 " NO ROOM"
174 PROMPT
175 GTO 18

176♦LBL "NNN"
177 CLA

178♦LBL 07
179 PROMPT
180 XEQ "D↑C"
181 RCL [
182 GTO 07

183♦LBL "D↑C"
184 INT
185 OCT
186 X=0?
187 GTO 06
188 STO ]
189 RDN
190 4 E2
191 ST+ ]
192 X<> ]
193 X<> d
194 FS?C 11
195 SF 12
196 FS?C 10
197 SF 11
198 FS?C 09
199 SF 10
200 FS? 07
201 SF 09
202 FS? 06
203 SF 08
204 SF 03
205 ARCL d
206 STO d
207 "⊢**"
208 CLX
209 STO ↑
210 X<> ]
211 X<> \
212 STO [
213 RDN
214 RTN

215♦LBL 06
216 "⊢♦"
217 STO ↑
218 STO ]
219 RDN
220 RTN

221♦LBL "CKA"
222♦LBL 22
223 XEQ 00

224♦LBL 08
225 CLX
226 X<> IND Z
227 SF 25
228 X=0?
229 FS?C 25
230 GTO 09
231 ISG Z
232 GTO 08

233♦LBL 09
234 RDN
235 STO c
236 CLX
237 STO '
238 STO e
239 BEEP
240 RTN

241♦LBL "SAX"
242 XEQ 10
243 RDN
244 X<>Y
245 X<> IND L
246 GTO 09

247♦LBL "RAX"
248 XEQ 10
249 RDN
250 RCL IND L
251 FC? 25
252 ENTER↑

253♦LBL 09
254 X<>Y
255 X<> c
256 RDN
257 FS?C 25
258 RTN
259 SF 99

260♦LBL 10
261 16
262 -
263 ABS
264 RDN
265 GTO 11

266♦LBL "C16"
267 XEQ 11
268 RDN
269 RTN
270♦LBL 00
271 XEQ "FEA"
272 193
273 -
274 X<0?
275 GTO 18
276 176
277 +
278 E3
279 /
280 176
281 +
282 ENTER↑

283♦LBL 11
284 XEQ 12
285 "α*i*"
286 X<> [
287 STO \
288 "⊢AB"
289 X<> \
290 ENTER↑
291 X<> c
292 X<>Y
293 SF 25
294 RTN

295♦LBL "GTE"
296 XEQ 12
297 X<> d
298 SF 02
299 SF 03
300 X<> d
301 CLA
302 STO [
303 "⊢AB"
304 X<> \
305 STO b

306♦LBL "FEA"
307 XEQ 12
308 X<> d
309 FS?C 07
310 SF 05
311 FS?C 08
312 SF 06
313 FS?C 09
314 SF 07
315 FS?C 10
316 SF 09
317 FS?C 11
318 SF 10
319 FS?C 12
320 SF 11
321 X<> d
322 DEC
323 RTN

324♦LBL 12
325 CLA
326 RCL c
327 X<> [
328 "⊢♦♦♦♦x̄"
329 X<> [
330 X<> d
331 CF 00
332 CF 01
333 CF 02
334 CF 03
335 X<> d
336 RTN

337♦LBL "PKA"
338 XEQ 00
339 CLA
340 CF 03

341♦LBL 20
342 FS?C 03
343 GTO 21
344 CLX
345 X<> IND Z
346 SF 25
347 X=0?
348 FS?C 25
349 GTO 21
350 ASTO a
351 CF 01
352 STO [
353 ASHF
354 X<> [
355 X=0?
356 SF 01
357 CLX
358 "⊢♦"
359 STO \
360 "⊢♦♦"
361 X<> \
362 "⊢♦"
363 X<> \
364 X=0?
365 GTO 13
366 X<> \
367 ASTO [
368 "⊢♦"
369 STO \
370 "⊢♦"
371 X<> \

372♦LBL 13
373 STO \
374 FC?C 01
375 "⊢♦♦♦"
376 X<> \
377 CLA
378 STO [
379 ASTO X
380 CLA
381 ARCL a
382 ARCL X
383 SF 03
384 ISG Z
385 CF 03
386 CLX
387 RCL \
388 X=0?
389 GTO 20
390 ASTO X
391 ASHF
392 ASTO a
393 ""
394 ARCL X
395 CLX
396 X<> [
397 STO IND T
398 ARCL a
399 ISG T
400 GTO 20
401 RTN

402♦LBL 21
403 X<> [
404 ""
405 X<> [
406 "⊢↑"
407 X<> \
408 "⊢↑↑↑↑↑↑"
409 X<> \
410 X<> IND L
411 SIGN
412 X≠0?
413 GTO 14
414 CLX
415 LASTX
416 ""
417 X<> [
418 "⊢↑"
419 X<> \
420 "⊢↑↑↑↑↑↑"
421 X<> \
422 STO IND T

423♦LBL 14
424 RDN
425 STO c
426 RDN
427 INT
428 176
429 -
430 X=0?
431 GTO 22
432 RDN
433 INT
434 175
435 -
436 BEEP
437 .END.
```

Routine Listing For: MK

```
01•LBL "MK"        73 STO Y            144 X<> c
02 CF 07           74  E1              145 RCL [
03 SF 09           75 ST/ Z            146 STO IND Z
04 12              76 MOD              147 X<>Y
05 XROM "VS"       77 8                148 X<> c
06 FC?C 25         78 *                149 CLST
07 PROMPT          79 ENTER↑           150 CLA
                   80 CF 08            151 FC? 10
08•LBL 01          81 LASTX            152 ISG 09
09 XROM "LF"       82 FS? 09           153 SF 20
10 STO 09          83 ST+ Y            154 FS? 07
11  E              84 R↑               155 RTN
12 +               85 INT              156 FS? 20
13 X<>Y            86 X≠0?             157 GTO 03
14 STO 10          87 X>Y?             158 "DONE, NO MORE"
15 ASTO 11         88 GTO 08           159 SF 09
16 DSE Y           89 R↑               160 GTO 14
17 GTO 07          90 +
18 SF 20           91 ST+ Z            161•LBL 07
19 FC?C 09         92 X<>Y             162 "NO ROOM"
20 GTO 13          93 X<=Y?            163 CF 20
                   94 CLX              164 CLST
21•LBL 02          95 X≠0?
22 RCL 09          96 SF 08            165•LBL 14
23 XEQ 11          97 +                166 FS?C 25
24 "REG FREE: "    98 36               167 RTN
25 RCL d           99 -                168 XROM "VA"
26 FIX 1           100 X>0?            169 TONE 7
27 ARCL Y          101 GTO 08          170 TONE 3
28 STO d           102 FC? 09          171 STOP
29 XROM "VA"       103 RCL ⁷           172 GTO 01
30 TONE 6          104 FS? 09
31 PSE             105 RCL e           173•LBL 08
                   106 FC? 08          174 "NO SUCH KEY"
32•LBL 03          107 GTO 14          175 GTO 14
33 "PRE↑POST↑KEY"  108 STO [
34 CLST            109 "⊢*"            176•LBL 09
35 XROM "VA"       110 X<> [           177 X<> d
36 TONE 7                              178 "KEY TAKEN"
37 STOP            111•LBL 14
38 GTO 14          112 X<> d           179•LBL 14
                   113 FS? IND Y       180 CF 09
42•LBL "+K"        114 GTO 09          181 CF 20
43•LBL 14          115 SF IND Y        182 FS?C 25
44 STO 08          116 X<> d           183 RTN
45 RDN             117 FC? 08          184 XROM "VA"
46 STO 07          118 GTO 14          185 TONE 3
47 RDN             119 STO [           186 PSE
48 STO 06          120 ARCL 10         187 "KEYCODE?"
49 CF 09           121 X<> \           188 CLST
50 RCL 10                              189 RCL 08
51 SIGN            122•LBL 14          190 XROM "VA"
52 FS? 20          123 FC? 09          191 TONE 7
53 X≠0?            124 STO ⁷           192 STOP
54 GTO 01          125 FS?C 09         193 STO 08
                   126 STO e           194 GTO 01
55•LBL 13          127 "**"
56 RCL 08          128 FS? 10          195•LBL "F?"
57 INT             129 ARCL 11         196 XROM "LF"
58 X=0?            130 X<> Z
59 FS? 07          131 RCL 07          197•LBL 11
60 FC?C 20         132 RCL 06          198 INT
61 GTO 02          133 XROM "DC"       199 LASTX
62 X<0?            134 XROM "DC"       200 FRC
63 SF 09           135 XROM "DC"       201  E3
64 ABS             136 FS?C 10         202 *
65 STO Z           137 GTO 14          203 X<>Y
66 44              138 "⊢***"          204 .5
67 -               139 ASTO 11         205 FC? 10
68 ABS             140 SF 10           206 SIGN
69 2                                   207 -
70 X<Y?            141•LBL 14          208 -
71 DSE T           142 RCL 09          209 END
72 R↑              143 RCL 10
```

TECHNICAL DETAILS

XROM: 10,01 — MK — SIZE: 012

Stack Usage: ALL CLEARED

0 T: USED
1 Z: PREFIX INPUT
2 Y: POSTFIX INPUT
3 X: KEYCODE INPUT
4 L: USED

Alpha Register Usage:

5 M:
6 N: ALL USED
7 O:
8 P:

Other Status Registers:

9 Q: NOT USED
10 ⊢: ALTERED
11 a: NOT USED
12 b: NOT USED
13 c: USED BUT RESTORED
14 d: USED BUT RESTORED
15 e: ALTERED

ΣREG: UNCHANGED

Data Registers:

R00: ONLY REGISTERS 6-11 ARE USED

R06: PREFIX
R07: POSTFIX
R08: KEYCODE
R09: INDEX FOR STORAGE
R10: Rc FOR LOWERED CURTAIN
R11: FIRST ASSIGNMENT OF PAIR

Flag Usage:

04: NOT USED
05: NOT USED
06: NOT USED
07: PROMPTLESS 1K
08: FLAG#>28 IN ⊢ OR E
09: SHOW#3FREE/SHIFTED KEY
10: 2ND ASSIGNMENT OF PAIR
20: REGISTER CHECK DONE

25: CLEARED

Display Mode: PRESERVED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 3

Global Labels Called:

Direct	Secondary
VS	E?
LF	OM
VA	2D
DC	PART OF GE

Local Labels In This Routine: LABEL 14 USED
01 REG. CHECK
02 "REG FREE" DISPLAY
03 INPUT PROMPT
07 NO ROOM
08 NO SUCH KEY
09 KEY TAKEN
11 CALCULATE # FREE
13 PROCESS KEYCODE

Execution Time: 7.4 - 12.4 sec. to first prompt (0-30 existing assignments; ≈6.2 sec. for each assignment.

Peripherals Required: NONE

Interruptible? NO
Execute Anytime? YES
Program File: MK
Bytes In RAM: 401
Registers To Copy: 61

Other Comments:

ML - MEMORY LOST RESIZE TO 017

After MASTER CLEAR is executed, there are 46 program registers available and a SIZE of 17+n*64 where n is the number of (single density) memory modules present. This SIZE is often insufficient for loading programs. ML provides a five-keystroke remedy. Executing ML immediately after MEMORY LOST will change the SIZE to 017 and 46+nx64 program registers will be available. XEQ ML requires half the keystrokes that SIZE 017 requires. As an extra bonus, the display is changed to FIX 2 for convenience.

COMPLETE INSTRUCTIONS FOR ML

WARNING - Use ML only immediately after MASTER CLEAR. If you don't have MEMORY LOST status, using ML will soon get you there.

1. Execute MASTER CLEAR (see page 242 of the *HP-41C OWNERS MANUAL*, step 3).
2. XEQ ML
3. You now have SIZE 017, FIX 2. The program mode display will still show 00 REG 46 even after packing. This is because MASTER CLEAR gives a packed .END. which ML does not disturb. ML only changes the curtain pointer, not the packed .END.
4. Reading a program card or pressing the back-arrow with .END. REG 46 in the display will change it to an unpacked .END. which can be PACKed. This step is not necessary, however, since for program entry purposes the 46+nx64 registers are immediately available for use.

LINE BY LINE ANALYSIS OF ML

Lines 01 - 03 determine the SIZE (17+n*64). Lines 04 - 09 convert this into 14+nx4, which represents (base 10) the first two digits needed for the new curtain pointer. Lines 10 - 13 attach the new curtain pointer to the other required information and store it in c. The ΣREG pointer is reset to 11, FIX 2 is selected, and the program pointer ends up at line 00 of user program memory.

CONTRIBUTORS HISTORY FOR ML

ML was conceived and written by Keith Jarett (4360) during the ROM loading.

FINAL REMARKS FOR ML

ML is somewhat of a frill in the PPC ROM, but it belongs to the class of programs which can only be used if they are in ROM. It was included because several bytes happened to be available for it in ROM 10.

FURTHER ASSISTANCE ON ML

Call Keith Jarett (4360) at (213) 374-2583.
Call Clifford Stern (4516) at (213) 748-0706.

Routine Listing For: ML

```
01♦LBL "ML"          15 FIX 2
02 17                16 XROM "GE"
03 XEQ 13            35♦LBL 13
04 16                36 64
05 -                 37 MOD
06 LASTX             38 SF 25
07 /
08 14                39♦LBL 02
09 +                 40 RCL IND X
10 "*i"              41 FC? 25
11 XROM "DC"         42 RTN
12 "⊦"               43 X<> L
13 ASTO c            44 +
14 ΣREG 11           45 GTO 02
```

TECHNICAL DETAILS

XROM: 10,12 — **ML** — SIZE: MASTER CLEAR

Stack Usage:
- 0 T: USED
- 1 Z: USED
- 2 Y: Y
- 3 X: X
- 4 L: USED

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: CLEARED
- 12 b: BYTES 1-5 CLEARED
- 13 c: ALTERED
- 14 d: NOT USED
- 15 e: NOT USED

ΣREG: UNCHANGED AT 11

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: MANY USED BUT ALL RESTORED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 0

Global Labels Called:

Direct	Secondary
DC	
GE	
part of S?	

Local Labels In This Routine:
NONE

Execution Time: 3.5 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO!
Program File:	ML
Bytes In RAM:	55
Registers To Copy:	64

Other Comments:
Execute only after MASTER CLEAR.

ML

APPENDIX H TABLE OF TABLES

NOTES

END

MP - MULTIPLE VARIABLE PLOT (1-9)

This routine generates plots on the 82143A printer in standard resolution (1 plot point per printed line) for between 1 and 9 functions or sets of tabular values, simultaneously. The resolution of MP plots in the X direction (the long dimension of the printer paper) is the same as for the printer's PRPLOT routine. (For high resolution plotting, see the HP routine writeup on page 188.) Each plot symbol consists of a single column of thermal print dots, and can fall in any of the 168 columns across the printed line.

MP has the following features:

- A. Function plotting or tabular value plotting
- B. Variable plot width (1 to 168 columns)
- C. Initial header information either printed or suppressed
- D. Standard Y-axis (12 double-width dashes) printed at the beginning and end of each function plot, or replaced by a pair of user-defined axes
- E. Completely adjustable plot symbol usage and order
- F. Two user-definable plot symbols (of 9 total)
- G. 4 different overflow modes selectable for cases where values exceed Y min or Ymax
- H. X axis (or axes) printed in any selected column(s).

In addition, the following topics will also be discussed:

- I. Prompting for user inputs to MP
- J. Advanced MP applications

Each of the above options and features will be explained and illustrated through detailed instructions and various plot examples that follow, with the above list used as an outline for the main text. Four basic examples that illustrate the principal modes of operation for the MP routine are presented first.

Basic Single-Function Plotting Example:

Example 1. Use MP to plot a sine function over one full period of 0 to 360 degrees in increments of 15 degrees. Use the full plot width of 168 columns. Let the Y limits be -1 and 1. Label the function 'SINE' and store the label in the appropriate register. The keystroke sequence is as follows:

KEYSTROKES	DISPLAY	RESULT
GTO . .	PACKING, etc.	
SIZE 024		
PRGM LBL ALPHA		Global label of
SINE ALPHA	01 LBL SINE	function to plot
SIN	02 SIN	Take sine of X

KEYSTROKES	DISPLAY	RESULT
RTN PRGM	03 RTN	Exit here
XEQ DEG	0.0000	Degrees mode
XEQ RF	0.00	Clear user flags
-1 STO 00	-1.00	Store Y minimum
CHS STO 01	1.00	Store Y maximum
168 STO 02	168.00	Store plot width
0 STO 08	0.00	Store X minimum
360 STO 09	360.00	Store X maximum
15 STO 10	15.00	Store X increment
ALPHA SINE	SINE	Name of function
ASTO 15	SINE	Store in R15
ALPHA 1	1	No. of functions
XEQ MP	---	Plots function

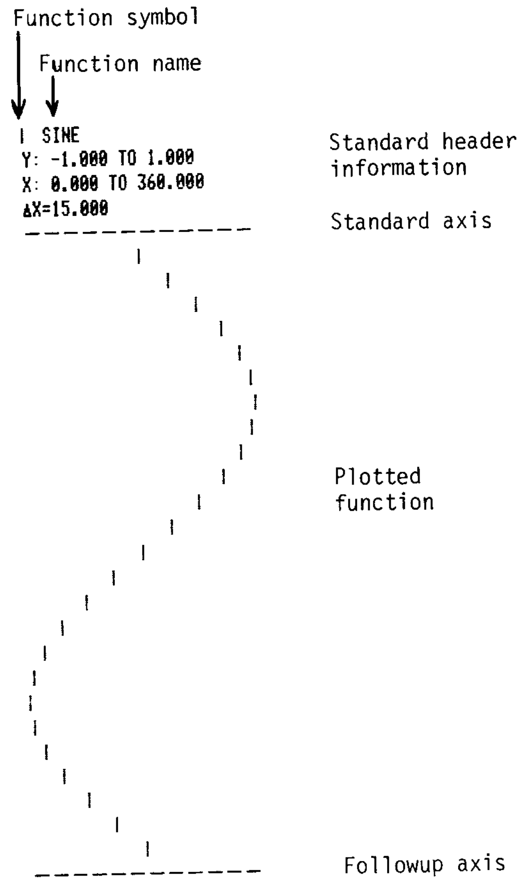


Figure 1. The sine curve plotted using MP in Example 1. Execution time: 2 min 7 sec.

Basic Single-Column Value-Plotting Example:

Example 2. Use MP to plot the data in Table 1.

Point #	Value
1	5
2	12
3	18
4	20
5	26
6	21
7	22
8	27
9	18
10	8

Table 1. Tabular values to be plotted in Example 2.

Let the Y-direction limits be 0 and 30, and make the plot 168 columns wide.

The required keystrokes are:

KEYSTROKES	DISPLAY	RESULT
SIZE 024	(Orig. X reg.)	
XEQ RF	0.00	Clears F00-F28
0		
STO 00	0.00	Y minimum
30		
STO 01	30.00	Y maximum
168		
STO 02	168.00	Plot width
SF10	168.00	Value plotting mode
5		
STO 15	5.00	1st plot value
1	1	# simultaneous values
XEQ MP		Plots 1st value
12		
STO 15	12.00	2nd plot value
1	1	# simultaneous values
XEQ MP		Plots 2nd value
18		
STO 15	18.00	3rd plot value
1		# simultaneous values
XEQ MP		Plots 3rd value
.	.	.
.	.	.
.	.	.
8		
STO 15	8.00	Last plot value
1	1	# simultaneous values
XEQ MP		Plots last value

The plot is shown in Figure 2.

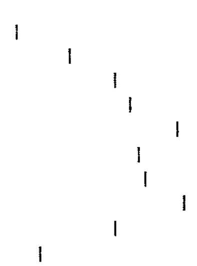


Figure 2. Plot of values in Table 1 using value-plotting mode of the MP routine. Note that plots of tabular values do not print any header information or axes.

Basic Multiple Function-Plotting Example:

Example 3. Use MP to plot the 5 functions Y=X↑0.3, Y=X↑0.6, Y=X↑0.9, Y=X↑1.2, and y=X↑1.5 simultaneously. Let Y limits be 0 and 20, X limits 0 and 10 with increments of 1. Use a plot width of 150 columns.

Name these 5 functions 'X3', 'X6', 'X9', 'X12' and 'X15'. Use the RF (Reset Flags) routine to clear any user flags and then set flags 21 and 55 to enable the printer. To set system flag 55, place 55 in X and call the IF (Invert Flags) routine in the ROM. The main program will be 'MPLT-6', listed here:

APPLICATION PROGRAM FOR: MP	
01♦LBL "X3" 02 .3 03 Y↑X 04 RTN	Function 1 of 5
05♦LBL "X6" 06 .6 07 Y↑X 08 RTN	Function 2 of 5
09♦LBL "X9" 10 .9 11 Y↑X 12 RTN	Function 3 of 5
13♦LBL "X12" 14 1.2 15 Y↑X 16 RTN	Function 4 of 5
17♦LBL "X15" 18 1.5 19 Y↑X 20 RTN	Function 5 of 5
21♦LBL "MPLT-6"	Function plotting routine
22 XROM "RF" 23 SF 21 24 55 25 XROM "IF"	Clear F00 to F28, SF21, SF55
26 0 27 STO 00	Ymin
28 20 29 STO 01	Ymax
30 150 31 STO 02	Plot width
32 0 33 STO 08	Xmin
34 10 35 STO 09	Xmax
36 1 37 STO 10	X increment
38 "X3" 39 ASTO 15	1st function name
40 "X6" 41 ASTO 16	2nd function name
42 "X9" 43 ASTO 17	3rd function name
44 "X12" 45 ASTO 18	4th function name
46 "X15" 47 ASTO 19	5th function name
48 5 49 XROM "MP" 50 END	# of functions to be plotted

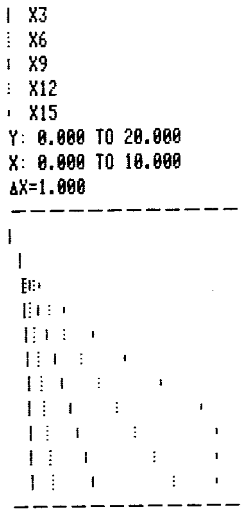


Figure 3. Five functions from Example 3 plotted simultaneously using MP function-plotting mode. Execution time: 5 min 0 sec.

Note that function #5, Y=X↑1.5 is 'clipped' at the upper Y limit (column 150 or Y=20) in the above example, since Y actually equals 31.62 at X=10. Such overflow situations are described in detail in a later section.

Basic Multiple-Column Value-Plotting Example:

Example 4. Use MP to plot the data in table 2.

Row #	1st Value	2nd Value	3rd Value
1	40	60	70
2	45	66	79
3	43	68	82
4	48	61	77
5	49	58	74
6	52	57	71
7	54	56	69
8	57	55	66

Table 2. Data to be plotted in Example 4.

This requires the value-plotting mode of MP. Let the Y limits be 30 and 80, with a plot width of the full 168 columns. The keystroke sequence shall be:

KEYSTROKES	DISPLAY	RESULT
SIZE 027		
XEQ RF	X	Clears F00-28
30	30	
STO 00	30.0000	Y minimum
80	80	
STO 01	80.0000	Y maximum
168	168	
STO 02	168.0000	Plot width
SF 10	168.0000	Value plotting mode

KEYSTROKES	DISPLAY	RESULT
40	40	
STO 15	40.0000	1st plot value
60	60	
STO 16	60.0000	2nd plot value
70	70	
STO 17	70.0000	3rd plot value
3	3	Number values plotted simultaneously
XEQ MP		Plots first line
45		
STO 15	45.0000	1st plot value
66		
STO 16	66.0000	2nd plot value
79		
STO 17	79.0000	3rd plot value
3	3	Number values plotted simultaneously
XEQ MP		Plots second line
.	.	.
.	.	.
.	.	.
57		
STO 15	57.0000	1st plot value
55		
STO 16	55.0000	2nd plot value
66		
STO 17	66.0000	3rd plot value
3	3	Number values plotted simultaneously
XEQ MP		Plots last line

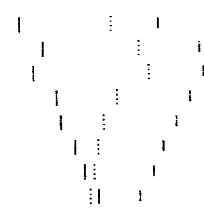


Figure 4. Plot of 3 sets of values from Table 2. Once again, note that no header or axes are printed when MP is in value-plotting mode.

A. COMPLETE INSTRUCTIONS FOR MP

A.1. For Function Plotting:

Load the following registers with the required input information:

R00 = Y minimum value (Left edge of plot, with Y direction across narrow width of printer paper)
R01 = Y maximum value (Right edge of plot)
R02 = Plot width (1 to 168 columns)
R04 = Global label of Y axis plotting routine in RAM (6 char's or less), with F09 set if used
R08 = X minimum value (First line of plot, with X direction down long dimension of printer paper)
R09 = X maximum value (Last line of plot)
R10 = X increment (delta X) value

R12 = Symbol/function map (stored only if symbol order/usage is changed and F04 set; default order is 0.123456789)
R15 = Global LBL of function #1 (6 char's or less)
R16 = Global LBL of function #2, if used
R17 = Global LBL of function #3, if used

. .
. .
. .

R23 = Global LBL of function #9, if used
R33 = User defined symbol #8, if used (= ACCOL #/1000, .001 to .127)
R34 = User defined symbol #9, if used (= ACCOL #/1000, .001 to .127)

Registers 03,05-07,11,13,14,24-32 are also used.

Registers 24 to 32 act as a 'software print buffer' to store the sorted column positions of the plot symbols. These registers are filled counting from R24 up for the number of functions to be plotted simultaneously. For example, if 5 functions are plotted, R24 to R28 are used. If only 1 function is plotted no sorting is performed and only R24 is used. Therefore, if no user-defined symbols are used, the minimum SIZE required is to the last register in the software print buffer. Registers 33 and 34 are required only if special user-defined symbols are used. In other words, actual register usage is R00 to R23 + one register per function and R33/34 if 1 or 2 user symbols are used.

Note that these are program assigned symbols for up to 7 functions. The user must define additional symbols if 8 or 9 functions are to be plotted.

A.1.1. Restrictions on functions plotted.

The functions in RAM which are plotted must accept input passed to them from MP in the X register and exit with output also in the X register. Global labels must not exceed 6 characters in length. The functions must not change the display mode without restoring it to FIX 0 (the mode when each function is called by MP) before returning.

A.1.2. Flag Usage for Function Plotting:

F10: Clear for function-plotting
F09: Set if user-defined Y-axis routine is used with global label in R04;
Clear if standard Y axis (12 dashes) is desired
F08: Used Internally
F07: Set if user wishes to skip standard header information (function symbols, Y limits printed);
Clear for standard header to be printed
F06 & F05 Plot Overflow Modes:

F06:	Clear	Set	Set	Clear
F05:	Clear	Clear	Set	Set
	points stay at edge of plot	points disappear at edge	points reflected back from the edge	points return from edge
	clipping mode	disappearing mode	mirror plot mode	wrap-around mode

F04: Set only if symbol order/usage is changed by the user (by storing symbol map in R12;
Clear otherwise

Now place the number of functions to be plotted simultaneously into the X register and XEQ MP to plot the functions from X minimum to X maximum.

A.2. For Value-Plotting:

Load the following registers with the required input:

R00 = Y minimum value
R01 = Y maximum value
R02 = Plot width (1 to 168 columns)
R12 = Symbol/function map (stored only if symbol order/usage is changed and F04 set)
R15 = Value #1 to be plotted
R16 = Value #2 to be plotted at the same time
R17 = Value #3 to be plotted at the same time

. .
. .
. .

R23 = Value #9 to be plotted at the same time
R33 = User defined symbol #8, if used (= ACCOL#/1000, .001 to .127)
R34 = User defined symbol #9, if used (= ACCOL#/1000, .001 to .127)

Maximum register usage is R00 to R34 with R24 to R32 used as a software print buffer as in function-plotting. The minimum required SIZE is determined the same way as for function-plotting: SIZE to accommodate up to R23 plus the number of values to be plotted simultaneously, and R33 and R34 if 1 or 2 user-defined symbols are used. (More on user-defined symbols later.)

A.2.1. Flag Usage for Value Plotting:

F10: Set for value-plotting
F09: Used internally
F08: Used internally
F07: Used internally
F06 & F05 Plot Overflow Modes: Use in same way as for function plotting.
F04: Set only if symbol order/usage is changed by user (by storing symbol map in R12)

Now place the number of values to be plotted simultaneously in the X register and XEQ MP to plot a line of values. Then load registers R15 and up with the next set of values, the number of values into X, XEQ MP , and the next line will be plotted, etc. Note that no header or Y axis is printed with value-plotting. Also, there is no Xmin, Xmax or X increment, since each printed line consists presumably of input from tabular data.

A.3. Execution Times for MP :

Running times for the examples presented were obtained by timing the actual programs using the ROM and printer. Speed of the HP41C/CV was obtained by executing the following short routine:

```
LBL 01   +   GTO 01
```

This routine was run beginning with 1 in the Y, Z, and T registers and with X clear. R/S was pressed, and then pressed again after 100 seconds to establish a speed count. Results ranged from the low 1600's to middle 1700's for various 41C's, so 1700 was established as a reference count. Execution times presented for each example have been normalized to the 1700 speed count.

The following relationship was obtained for the **MP** routine, using nonlinear regression analysis:

Execution
time, min = .02516-.02144*L + .09566*L*F

where L = Number of printed lines in plot,and
F = Number of functions plotted

This relationship holds for a 1700-count HP41C. Program MPT has been provided for the estimation of run times for **MP** plots, due to the wide range of times possible. This program will calculate estimated run times normalized to any count in the 100-second test above and then executes **MP** . If the speed count is not known for the particular 41C being used, then simply pressing R/S at the appropriate time will assume a reference count of 1700.

Enter parameters for **MP** into data registers, including the number of functions in X, and then:

KEYSTROKES	DISPLAY	RESULT
XEQ MPT	COUNT?	Prompts for count
Enter count, or just press R/S for 1700-count time		Prints 'EST RUN TIME:' and time, then runs **MP**

The listing for program MPT:

APPLICATION PROGRAM FOR: **MP**

Program	Comments
01♦LBL "MPT" 02 SF 08 03 GTO 00 04♦LBL "HPT" 05 CF 08	
06♦LBL 00 07 STO 03	Store # functions plotted in R03
08 1700 09 "COUNT?" 10 PROMPT 11 STO 04	Store 1700 or count in R04
12 RCL 09 13 RCL 08 14 - 15 RCL 10 16 / 17 1 18 + 19 FS? 08 20 GTO 01 21 11 22 /	Compute the number of lines to be plotted
23♦LBL 01 24 RCL X 25 RCL 03 26 * 27 FS? 08 28 .09566 29 FC? 08 30 .3615 31 * 32 X<>Y 33 FS? 08 34 -.02144 35 FC? 08 36 .1952 37 * 38 + 39 FS? 08 40 .02516 41 FC? 08 42 -.8905 43 + 44 1700 45 * 46 RCL 04 47 /	Calculate estimated run time for **MP** or for **HP**
48 "EST RUN TIME: 49 FIX 2 50 ARCL X 51 "⊢ MIN." 52 PRA 53 RCL 03 54 FS? 08	Print run time
55 XROM "MP" 56 FC? 08 57 XROM "HP" 58 END	Call **MP** or **HP**

Note that program MPT is also applicable to ROM routine **HP** , and may be called by XEQ HPT to generate estimated run times for **HP** as well. (See the **HP** writeup, on page 188.)

The barcode for MPT/HPT appears in Appendix N.

A.4. Changing Display Annunciators.

As part of the operation of **MP** , flag 55 (the printer existence flag) is synthetically cleared using the **IF** routine in order to trick the calculator into assuming that no printer is present. This speeds up non-printing operations some 20 percent, which is significant in a plot that may take several minutes to complete. During the execution of the **IF** routine, the display annunciators may change, such as 'RAD' coming on, or flag annunciators going off. This situation will remain until the **MP** routine stops. If the user halts execution prematurely, the annunciators will return to their original configuration. This will also reset flag 55, since the printer will now be detected to be present. Pressing R/S to restart will eventually cause **MP** to detect that F55 is set, and again call the **IF** routine to clear it, and the annunciators will again change. No changes will have actually occurred to flags or to any modes.

B. Variable Plot Width.

The plot width in columns is stored by the user in register R02. This can vary from 1 to 168 columns. This feature was illustrated in Basic Example 3 and will be used extensively in many of the examples that follow.

C. Skip Standard Header.

If flag 07 is clear, a standard set of initial header lines is printed before the function plots (only with F10 clear). This consists of each function name and its corresponding plot symbol plus the limits in the Y and X directions along with the X increment value. Setting flag 07 causes MP to skip the header information entirely and just print the Y axis, whether it is ths standard 12 dashes or a user-defined axis (to be described later). This allows another header to be substituted and printed immediately before MP is called, if the user desires.

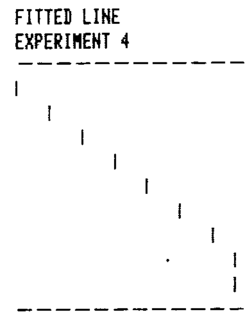


Figure 5. Plot from Example 5 using a custom header. Standard header has been skipped by setting flag 07 before calling MP . Program MPT predicted execution time to be .69 minutes, or 41 seconds, without printing the custom header. Actual execution time for this entire plot: 54 seconds.

MORE EXAMPLES OF MP

Example 5. Use MP to plot the function Y = 3X+4 with the new heading 'FITTED LINE' followed by 'EXPERIMENT 4'. Use limits Y=-5 and +5, X=-3 and +1, with delta X=0.5. Make the plot 160 columns wide.

APPLICATION PROGRAM FOR: MP	
01♦LBL "FL"	Function to be plotted
02 3	
03 *	
04 4	
05 +	
06 RTN	
07♦LBL "HDR"	Plotting routine
08 XROM "RF"	Clear user flags,
09 SF 21	SF21, SF55
10 55	
11 XROM "IF"	
12 -5	
13 STO 00	Ymin
14 CHS	
15 STO 01	Ymax
16 -3	
17 STO 08	Xmin
18 1	
19 STO 09	Xmax
20 .5	
21 STO 10	X increment
22 160	
23 STO 02	Plot width
24 "FL"	
25 ASTO 15	Function name
26 SF 07	Skip standard header
27 "FITTED LINE"	
28 PRA	
29 "EXPERIMENT 4"	New header lines
30 PRA	
31 1	# functions plotted
32 XROM "MP"	Call to MP
33 END	

FURTHER DISCUSSION OF MP

D. Custom User-Defined Y Axis and Axis Labels

The MP routine prints a pair of standard Y axes before and after the plotted functions if flag 09 is clear. These axes consist of 12 double-wide dashes spanning the full 24 character paper width. If the user wishes his own custom Y axis, he may write an axis program in RAM, store its global label (not more than 6 characters long) in R04, and set F09 for it to be printed. This axis would print before and after the plot.

D.1. Y Axis Numeric Labelling.

Numeric labels can be provided for user-defined Y axis by having the custom Y axis program print two rows; one for the labels and one for the axis. If numeric labelling of the Y axis is included and is desired only before the plot, a suitable test has to be provided in the user axis program.

Example 6. Plot the same function as in Example 5 but add a user-defined Y axis with tic marks at -5, -2.5, 0, 2.5 and 5, and label these tic marks numerically as well.

APPLICATION PROGRAM FOR: MP	
01♦LBL "FL"	Function to be plotted
02 3	
03 *	
04 4	
05 +	
06 RTN	
07♦LBL "HDR"	Plotting routine
08 XROM "RF"	Clear user flags,
09 SF 21	SF21, SF55
10 55	
11 XROM "IF"	

12 -5	
13 STO 00	Ymin
14 CHS	
15 STO 01	Ymax
16 -3	
17 STO 08	Xmin
18 1	
19 STO 09	Xmax
20 .5	
21 STO 10	X increment
22 160	
23 STO 02	Plot width
24 "FL"	Function name
25 ASTO 15	Skip standard header
26 SF 07	via SF07
27 "Y-AXIS"	Name of user-defined
28 ASTO 04	Y axis routine, skip
29 SF 09	standard axis (SF09)
30 "FITTED LINE"	New header lines
31 PRA	
32 "EXPERIMENT 4"	
33 PRA	
34 1	# functions plotted
35 XROM "MP"	Call to MP
36 END	
01•LBL "Y-AXIS"	Y axis printing RAM
02 FS? 00	routine
03 GTO 00	Skip numeric label-
04 CF 12	ling if F00 is set
05 "-5 -2.5 0"	Y direction numeric
06 "├ 2.5 5"	labelling
07 PRA	
08•LBL 00	
09 CF 12	Y axis
10 127	
11 ACCOL	printing
12 SF 12	
13 "---"	section
14 ACA	
15 CF 12	
16 10	
17 SKPCOL	
18 X<>Y	
19 ACCOL	
20 SF 12	
21 ACA	
22 CF 12	
23 11	
24 SKPCOL	
25 127	
26 ACCOL	
27 SF 12	
28 ACA	
29 X<>Y	
30 CF 12	
31 SKPCOL	
32 X<>Y	
33 ACCOL	
34 SF 12	
35 ACA	
36 X<>Y	
37 CF 12	
38 SKPCOL	
39 X<>Y	
40 ACCOL	
41 PRBUF	Set F00 to skip
42 SF 00	numeric labelling
43 END	when called again

```
FITTED LINE
EXPERIMENT 4
-5 -2.5   0    2.5   5
|-- |-- |-- |-- |
|
   |
      |
         |
            |
               |
                  |
                     |
                        |
                        |
|-- |-- |-- |-- |
```

Figure 6. Plot of example 6 with a custom Y axis and initial Y axis numeric labelling added by setting F09 and storing the axis routine global label 'Y-AXIS' in register 04. This routine checks the status of flag 00 (initially clear) in order to decide whether to skip the numeric labelling. Since labelling is not desired with the second printing of the axis, the routine sets F00 just before exiting the first time called. Execution time is 1 min 1 sec.

D.2. X Value Numeric Labelling.

Just as a user-defined custom Y axis and Y-directional labelling is possible, so is numeric labelling in the X direction. This is done by accumulating numeric information into the print buffer before any plot symbols are placed there. The simplest way to do this is to have function #1 of a multi-function plot place each X label into the print buffer before exiting with its function value in the X register. Of course, the plot width must decrease to allow space for these labels. If the plot width added to the label width exceeds 168 columns, MP will plot it anyway, but buffer overflow will result.

D.2.1. X-Axis Labelling Using ACX.

There are various ways that the X labels can be placed into the print buffer. The first way is to use the printer's ACX instruction. This requires that the same number of characters are accumulated for each X label, regardless of the label's number of digits. In addition, an extra printer character position must be allotted for the sign of the X label when ACX is used, even if all the labels are positive. To assure that an equal number of printer positions is used for all labels, a test using the log of the X value may be implemented. This will yield the correct number of blank spaces which have to be skipped ahead of a label that is shorter than the maximum possible anticipated length. For example, if the maximum number of digits to the left of the decimal point in the labels is 3, then the following log test can assure an equal number of printer positions:

X label	Skip amount prior to adding label: 2-INT(LOG(ABS(X)))		
101	2-	2	= 0 spaces
-50	2-	1	= 1 space
8	2-	0	= 2 spaces

Example 7. Use MP to plot Y =(X/100)↑2 for X= -250 to +250 with delta X= 50. Let Y limits be 0 and 7. Include X labels using ACX in FIX 0 display mode.

Instead of having the X-labelling be an integral part of the function to be plotted, call it as a separate subroutine, XLA. This labelling program must also test for a zero X value, since the log of zero will cause an error:

APPLICATION PROGRAM FOR: MP	
01♦LBL "X2/100"	Function plotted
02 XEQ "XLA"	Calls X label routine
03 100	
04 /	
05 X↑2	
06 1	
07 SKPCHR	
08 RDN	
09 RTN	
10♦LBL "XLA"	X labelling routine
11 ENTER↑	
12 ABS	
13 X=0?	Test for zero; re-
14 1	place with 1
15 LOG	
16 INT	Calculates # spaces
17 CHS	to skip before
18 2	accumulating X label
19 +	into print buffer
20 SKPCHR	
21 X<>Y	
22 ACX	
23 RTN	
24♦LBL "PLT"	Function plotting
25 XROM "RF"	routine
26 SF 21	Clear user flags,
27 CF 29	SF21, SF55
28 55	
29 XROM "IF"	
30 0	
31 STO 00	Ymin
32 7	
33 STO 01	Ymax
34 133	
35 STO 02	Plot width
36 -250	
37 STO 08	Xmin
38 CHS	
39 STO 09	Xmax
40 50	
41 STO 10	X increment
42 "X2/100"	Function name
43 ASTO 15	
44 1	No. functions plotted
45 XROM "MP"	Call to MP
46 END	

```
I X2/100
Y: 0.000 TO 7.000
X: -250.000 TO 250.000
ΔX=50.000
------------
-250          I
-200        I
-150      I
-100    I
 -50 I
   0 I
  50 I
 100    I
 150      I
 200        I
 250          I
------------
```

Figure 7. Plot of Y=(X/100)↑2 from Example 7, with X-direction labelling for each printed line. The labelling routine XLA, called by the plotted function, accumulates values into the print buffer using the ACX instruction. Execution time: 1 min 9 sec.

D.2.2. X-Axis Labelling Using CP .

Another way to label the X direction is to use the ROM routine CP (Column Print Formatting) to align the column of numeric X labels. (See page 98 for the CP instructions.) The routine XLA from Example 7 could be modified as follows to produce the same plot:

10♦LBL "XLB"	X labelling routine
11 STO 25	Save X in R25
12 RCL 06	
13 STO 26	Save R06 in R26
14 2	
15 STO 06	Set CP skip index
16 RCL 25	
17 XROM "CP"	Call CP for X label
18 RCL 26	
19 STO 06	Restore R06 from R26
20 RCL 25	Restore X from R25
21 RTN	

Both CP and MP use register R06, so the original value was temporarily stored in R26 while CP was called. In addition, the X value was saved in R25 until CP was finished, so it could be recalled and used in function X2/100.

D.2.3. X-Axis Labelling Using ACA.

A third approach to X-direction labelling is the use of the ACA printer function, printing X labels directly from ALPHA. The advantage of using ACA is that X labels do not require an extra blank printer position skipped for a negative sign. This obviously will allow for a wider plot field. Since MP already uses the M,N and O registers for numeric counters, however, these must be preserved during the ACA process. One solution is to store M, N and O to 3 consecutive data registers and recall them back later. In example 8 below, the 3 registers are saved in stack positions X, Y and Z, with the X value pushed up to T.

Example 8. Use MP to plot Y=X↑2 for X from 0 to 9, delta X of 1; Y from 0 to 100 and a plot width of 168 columns. Label the X direction using the ACA function.

APPLICATION PROGRAM FOR: MP	
01◆LBL "X2" 02 RCL [03 RCL \ 04 RCL] 05 FIX 0 06 CLA 07 ARCL T 08 ACA 09 STO] 10 RDN 11 STO \ 12 RDN 13 STO [14 RDN 15 1 16 SKPCHR 17 RDN 18 X↑2 19 RTN	Function plotted Save M, N & O in X, Y and Z Put value into buffer Restore M, N & O Skip 1 character Compute the square
20◆LBL "PT" 21 XROM "RF" 22 SF 21 23 55 24 XROM "IF" 25 CF 29 26 0 27 STO 00 28 100 29 STO 01 30 154 31 STO 02 32 0 33 STO 08 34 9 35 STO 09 36 1 37 STO 10 38 "X2" 39 ASTO 15 40 1 41 XROM "MP" 42 END	Function plotting routine Clear user flags, SF21, SF55, CF29 Ymin Ymax Plot width Xmin Xmax X increment Function name # functions plotted Call to MP

```
| X2
Y: 0.000 TO 100.000
X: 0.000 TO 9.000
ΔX=1.000
------------
0 |
1 |
2  |
3   |
4    |
5      |
6        |
7          |
8             |
9                |
------------
```

Figure 8. Plot of Y=X↑2 from Example 8. Labelling of the X direction is achieved using the ACA printer instruction, which does not require a leading blank printer character for negative signs when the value accumulated is positive. Execution time: 59 sec.

E. Standard Symbols, Symbol Order and Symbol Precedence.

The standard set of 7 one-column plot symbols for MP is shown in Figure 9.

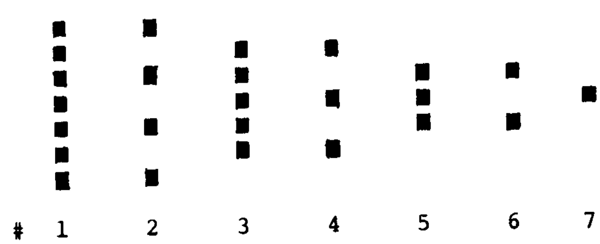


Figure 9. The 7 standard 1-column plot symbols for the MP routine. Each '■' represents a filled thermal dot in the 7 dot column.

E.1. Remapping Symbol Order/Changing Symbol Usage.

If MP plot function names are simply placed in the appropriate registers, the plotted functions will use the plot symbols in the order shown in Figure 9. Symbol order and usage are entirely controllable by the user however. Register R12 contains the 'symbol map'- a decimal number that matches the Nth digit to the Nth function to be plotted. If R12 isn't specifiedby the user, then MP sets it to a value of 0.123456789 - the standard order of symbols. This can easily be changed, however, before executing the MP routine. For instance, if 4 functions are to be plotted using the first 4 symbols in reverse order, merely set flag F04 and store .4321 into R12 before executing MP. If 6 functions are all to use symbol #7, then store .777777 in R12, SF04 and XEQ MP .

Example 9. Use MP to plot the graph in Example 3, but use symbols 7,5,3,1 and 4 for functions 1 to 5 respectively.

APPLICATION PROGRAM FOR:	MP
21♦LBL "MPLT-6"	Function plotting routine
22 XROM "RF"	
23 SF 21	Clear user flags, SF21, SF55
24 55	
25 XROM "IF"	
26 0	
27 STO 00	Ymin
28 20	
29 STO 01	Ymax
30 150	
31 STO 02	Plot width
32 0	
33 STO 08	Xmin
34 10	
35 STO 09	Xmax
36 1	
37 STO 10	X increment
38 "X3"	
39 ASTO 15	1st function name
40 "X6"	
41 ASTO 16	2nd function name
42 "X9"	
43 ASTO 17	3rd function name
44 "X12"	
45 ASTO 18	4th function name
46 "X15"	
47 ASTO 19	5th function name
48 .75314	
49 STO 12	Remap 1st 5 symbols
50 SF 04	Calls new R12 usage
51 5	# functions plotted
52 XROM "MP"	Call to MP
53 END	

```
· X3
ı X6
ı X9
| X12
: X15
Y: 0.000 TO 20.000
X: 0.000 TO 10.000
ΔX=1.000
```

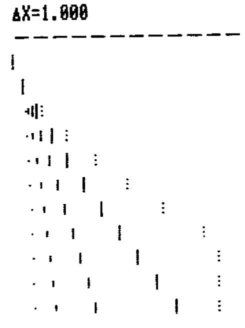


Figure 10. Five functions from Example 3 plotted simultaneously with symbol usage changed, as specified by the value .75314, stored in register R12. Flag 04 is set here also for MP to use the new symbol map. Execution time: 5 min 1 sec.

E.2. Symbol Overlap Precedence.

When two functions occupy the same column, the function using the lower numbered symbol will always be plotted, masking the other symbol. Thus symbol #1 will always appear, while #2 will mask symbols 3 through 7, etc. The program checks the value of the equivalent ACCOL number corresponding to the plot symbols, and always plots the higher valued ACCOL number. Symbols 1 through 7 have ACCOL numbers in decreasing order, causing the precedence rule described here. Note in Example 9 above (Figure 10), symbol #1 was printed for the first two lines of the plot, even though all 5 functions occupied the same column in both of these lines.

F. User-Defined Plot Symbols.

In addition to the standard set of single column plot symbols described above, MP also permits 2 user-definable single column symbols to be specified. These are numbered 8 and 9 in the routine for use in the symbol map in R12. To define symbols 8 and/or 9, store the ACCOL number of the desired column symbol, divided by 1000, into R33 for #8 and R34 for #9. The values stored in these registers would thus range from .001 to .127 inclusive. These two symbols are flexible as far as overlap precedence over the standard symbols is concerned, since the ACCOL numbers can vary. The rule is the same with these symbols - when 2 functions occupy the same column, the one with the higher ACCOL number will be the one that is plotted, while the other one will be concealed. The ACCOL numbers for the standard symbols are shown below:

SYMBOL #	1	2	3	4	5	6	7
ACCOL #	127	85	62	42	28	20	8

Example 10. Use MP to plot Y=X↑2, Y=4 and Y=4X/3 simultaneously. Y limits: 0 and 25; X limits 1 and 5, delta X=0.5. Use symbol #8 as ACCOL #99 for function #1, symbol #2 for function #2 and symbol #9 as ACCOL #65 for function #3. Use standard header and Y axis, label the X direction using FIX 1 display mode.

APPLICATION PROGRAM FOR:	MP
01♦LBL "X-2" 02 FIX 1 03 ACX 04 1 05 SKPCHR 06 RDN 07 X↑2 08 FIX 0 09 RTN	Function #1 X labelling via ACX
10♦LBL "TH" 11 4 12 RTN	Function #2
13♦LBL "TW" 14 1.3333 15 * 16 RTN	Function #3
17♦LBL "MPLT-9" 18 XROM "RF"	Function plotting routine
19 SF 21 20 55 21 XROM "IF"	Clear user flags, SF21, SF55
22 0 23 STO 00	Ymin
24 25 25 STO 01	Ymax
26 133 27 STO 02	Plot width
28 1 29 STO 08	Xmin
30 5 31 STO 09	Xmax
32 .5 33 STO 10	X increment
34 "X-2" 35 ASTO 15	1st function name
36 "TH" 37 ASTO 16	2nd function name
38 "TW" 39 ASTO 17	3rd function name
40 .829 41 STO 12	Remap 1st 3 symbols
42 SF 04 43 .099 44 STO 33	User symbol #8
45 .065 46 STO 34	User symbol #9
47 3	# functions plotted
48 XROM "MP" 49 END	Call to MP

```
: X-2
⁝ TH
: TW
Y: 0.000 TO 25.000
X: 1.000 TO 5.000
ΔX=0.500
```

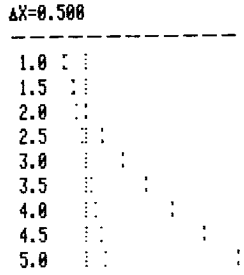


Figure 11. Three functions of Example 10 plotted simultaneously using MP with special symbols. Note in the above example, function #1 (using symbol #8, ACCOL #99) takes precedence over function #2 (standard symbol #2, ACCOL #85) at the plot line corresponding to X = 2.0. In the line for X=3.0, however, symbol #2 prevails over symbol #9 (ACCOL #65). Execution time: 2 min 24 sec.

G. *'Overflow' Modes.*

In the standard PRPLOT routine of the 82143A printer, if the value of the plotted function exceeds either the Y minimum or maximum limits specified by the user, the plot symbol is printed at the edge of the plot field. This tells the user nothing about the function beyond the limits of the printer paper, except that the points lie beyond the edge. In MP, the user has 4 options to choose from in dealing with function overflow. These options are controlled by the 4 combined states of flags 05 and 06, as described below:

F05 CLEAR, F06 CLEAR:
Function points remain at the edge of the print field, when overflow occurs ('Clipping Mode').

F05 CLEAR, F06 SET:
Function points disappear at the edge that they exceed ('Disappearing Mode').

F05 SET, F06 SET:
Function points are reflected back from the edge they exceed ('Mirror Plotting').

F05 SET, F06 CLEAR:
Function points wrap around and return onto the plot field from the opposite edge that they exceed ('Wraparound Plotting').

The way each of the 4 overflow modes behaves when a function exceeds the user-specified limits is shown in Figure 12. The following program was used to generate the 4 plots in Figure 12. The only change made for each plot was the state of flags F05 and F06. The Y limits were purposely chosen to be too narrow to fit the full range of the function plotted.

01♦LBL "S" 02 SIN 03 RTN	Function plotted
04♦LBL "S-4" 05 -.8 06 STO 00	Function plotting routine Ymin
07 CHS 08 STO 01	Ymax
09 168 10 STO 02	Plot width
11 0 12 STO 08	Xmin
13 360 14 STO 09	Xmax

15 18	
16 STO 10	X increment
17 "S"	
18 ASTO 15	Function name
19 CF 05	
20 CF 06	1 of 4 overflow modes
21 1	# functions plotted
22 XROM "MP"	Call to MP
23 END	

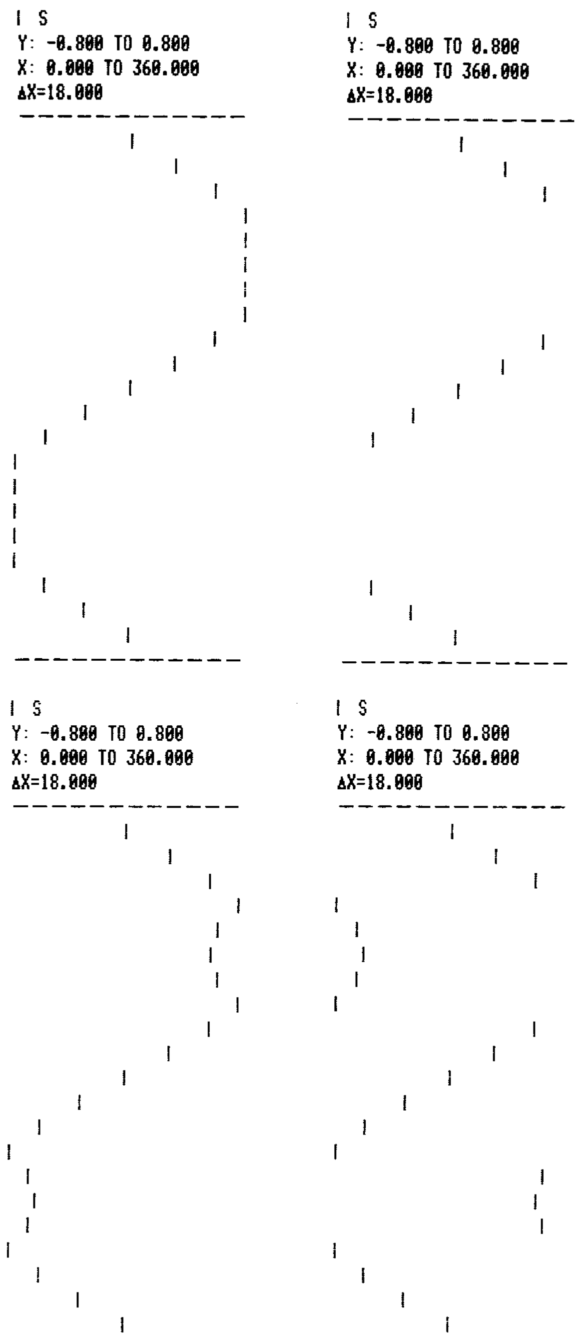


Figure 12. The 4 modes of overflow for the MP routine shown by a sine curve drawn with symbol #1 from 0 to 360 degrees in increments of 18 degrees. Upper left: plot is clipped at the edge; upper right: plot disappears at the edge; lower left: plot reflected at the edge; lower right: plot wrapped around to the opposite edge.

G.1. Mirror Plotting.

In one of the overflow modes, plotted points are reflected back from the edge which they exceed. This was originally called mirror plotting, and was submitted as a separate routine for the PPC ROM by Frits Kuyt (236). His idea was to have a short routine that could be called by the plotted function program to reflect the overflow points. It would be compatible with all the plotting routines in the ROM, and also work with the PRPLOT printer routine. His program is listed here:

01♦LBL "MR"	
02 RCL 01	Recall Ymax
03 X<>Y	
04 X<=Y?	
05 GTO 00	If value <Ymax,skip
06 -	
07 RCL 01	If not, reflect back
08 +	on upper edge
09♦LBL 00	
10 RCL 00	
11 X<>Y	
12 X>Y?	If value >Ymin, RTN
13 RTN	
14 -	
15 RCL 00	
16 +	If not, reflect back
17 END	on lower edge

Registers R00 and R01 contain the Y minimum and Y maximum values for the plot, which is true of PRPLOT, MP and HP (High Resolution Multifunction Plotting). While plotting functions using PRPLOT, if a function produces values that not only exceed a Y limit in a plot, but also exceed the opposite limit when reflected back it would be necessary to execute the mirror plotting routine several times in succession. Otherwise, if a function plotted by PRPLOT has already been reflected by MR exceeds the opposite edge of the plot, it will stay at that edge.

A single call of the MR routine will actually reflect values exceeding the upper Y limit as much as a second time if they must also be reflected back up from the lower edge. If a value exceeding the lower limit must be reflected a second time, however, it will stay at the edge of the plot. It is best, therefore, to call MR repeatedly to prevent this occurrence.

The version of mirror plotting built into the MP and HP routines will automatically reflect a function, no matter how many times it exceeds the plotting limits. This is illustrated in the following examples. Flags 05 and 06 are both set for this mode. (See also the HP routine writeup, elsewhere in this manual.)

Example 11. Plot the sine curve Y = 8 sin X from 180 to 360 degrees in 10 degree increments. Label the X direction. Y limits shall be -1 and 1. Plot the curve four ways: 1) In standard 'clipping' overflow mode (CF05, CF06); 2) In 'clipping' mode with the function calling MR once; 3) 'Clipping' mode with the function calling MR 2 consecutive times; and 4) In 'mirror plotting' mode (SF05, SF06) without the MR routine.

APPLICATION PROGRAM FOR: **MP**	
01♦LBL "8SINX"	Function plotted
02 STO 29	Save X in R29
03 RCL 06	
04 STO 30	Save R06 in R30
05 2	
06 STO 06	Skip index for CP
07 RCL Z	
08 XEQ "CP"	Call CP
09 1	
10 SKPCHR	Skip 1 character
11 RCL 30	
12 STO 06	Restore R06 from R30
13 RCL 29	Restore X from R29
14 SIN	
15 8	
16 *	
17 XEQ "MR"	XEQ MR 0, 1 or 2
18 END	times (1st 3 plots)
01♦LBL "PL8"	Function plotting
02 XROM "RF"	routine
03 SF 21	Clear user flags,
04 CF 29	SF21, CF29, SF55
05 55	
06 XROM "IF"	
07 -1	
08 STO 00	Ymin
09 CHS	
10 STO 01	Ymax
11 133	
12 STO 02	Plot width
13 180	
14 STO 08	Xmin
15 360	
16 STO 09	Xmax
17 10	
18 STO 10	X increment
19 CF 05	
20 CF 06	Set clipping mode
21 "8SINX"	(mirror plot mode
22 ASTO 15	for 4th plot)
23 1	# functions plotted
24 XROM "MP"	Call to MP
25 END	

```
| 8SINX
Y: -1.000 TO 1.000
X: 180.000 TO 360.000
ΔX=10.000
------------
180        |
190|
200|
210|
220|
230|
240|
250|
260|
270|
280|
290|
300|
310|
320|
330|
340|
350|
360        |
------------
```

```
| 8SINX
Y: -1.000 TO 1.000
X: 180.000 TO 360.000
ΔX=10.000
------------
180        |
190   |
200              |
210                |
220                |
230                |
240                |
250                |
260                |
270                |
280                |
290                |
300                |
310                |
320                |
330                |
340              |
350    |
360        |
------------
```

```
| 8SINX
Y: -1.000 TO 1.000
X: 180.000 TO 360.000
ΔX=10.000
------------
180        |
190   |
200              |
210        |
220 |
230         |
240               |
250                |
260                |
270                |
280                |
290                |
300               |
310         |
320 |
330        |
340              |
350    |
360        |
------------
```

```
| 8SINX
Y: -1.000 TO 1.000
X: 180.000 TO 360.000
ΔX=10.000
------------
180        |
190   |
200             |
210        |
220 |
230        |
240               |
250            |
260         |
270        |
280         |
290            |
300               |
310        |
320 |
330        |
340              |
350    |
360        |
------------
```

Figure 13. Four plots of Example 11. The 2nd and 3rd use the MP 'clipping' overflow mode in conjunction with the MR routine. The first 3 plots simulate the same situation as PRPLOT which only allows plotted function values to stay at the edge when they exceed the Y limit there. The 4th plot uses the MP 'mirror plotting' mode, which reflects the function each time necessary.

Example 12. Plot the curve Y = 5X from X = 0 to 15 in increments of .5 using PRPLOT. Call MR 2 times consecutively in the function. Let the Y limits be 0 and 10. Let the axis be at 0.

Program	Comments
01♦LBL "5X"	Function plotted
02 5	
03 *	
04 XEQ "MR"	
05 XEQ "MR"	XEQ MR twice
06 END	

KEYSTROKES	DISPLAY	RESULT
XEQ PRPLOT	NAME?	
shift 5 X R/S	Y MIN?	Inputs name
0 R/S	Y MAX?	Inputs Ymin
10 R/S	AXIS?	Inputs Ymax
0 R/S	X MIN?	Inputs axis
0 R/S	X MAX?	Inputs Xmin
15 R/S	X INC?	Inputs Xmax
.5 R/S	-----	Inputs Xinc, plots graph

```
        PLOT OF 5X
      X (UNITS= 1.) ↓
      Y (UNITS= 1.) →
        0.0        10.0
        0.0
        |-----------------|
   0.0*
   0.5:    *
   1.0:        *
   1.5:            *
   2.0:                *
   2.5:            *
   3.0:        *
   3.5:    *
   4.0*
   4.5:    *
   5.0:        *
   5.5:            *
   6.0:                *
   6.5:            *
   7.0:        *
   7.5:    *
   8.0*
   8.5:    *
   9.0:        *
   9.5:            *
  10.0:                *
  10.5:                *
  11.0:                *
  11.5:                *
  12.0:                *
  12.5:                *
  13.0:                *
  13.5:                *
  14.0:                *
  14.5:                *
  15.0:                *
```

Figure 14. PRPLOT plot of Y=5X which calls MR 2 consecutive times before returning the result (Example 12). This is an example of the use of the MR routine in the PRPLOT or PRPLOTP printer programs. Note that calling MR twice was not enough to reflect the function all the times required to prevent clipping, since the function grows without bound. Execution time: 2 min 28 sec.

G.2. 'Disappearing' Overflow Mode.

Sometimes it is desirable for plotted function values to disappear when they exceed the Y limits of the plot. This is obtained with MP by setting flag 06 and with flag 05 clear. An example of the usefulness of this mode is in examining a specific portion of a function for its behavior, without having the nonessential sections plotted.

Example 13. Plot the 3 functions Y=X↑3, Y = X↑3 +3*X and Y = X↑3+3*X↑2+3*X simultaneously using MP . Use Y limits -5 and 5. X limits shall be -4 and 4 with an increment of 0.5. Use symbols 1,2 and 3 respectively. Use 'disappearing' overflow mode.

APPLICATION PROGRAM FOR: MP	
01♦LBL "X3"	Function #1
02 FIX 1	
03 ACX	
04 1	
05 SKPCHR	X label using ACX
06 RDN	
07 3	
08 Y↑X	
09 FIX 0	
10 RTN	
11♦LBL "X-3"	Function #2
12 ENTER↑	
13 3	
14 Y↑X	
15 X<>Y	
16 3	
17 *	
18 +	
19 RTN	
20♦LBL "X-3-"	Function #3
21 ENTER↑	
22 RCL Y	
23 3	
24 Y↑X	
25 X<>Y	
26 X↑2	
27 3	
28 *	
29 +	
30 X<>Y	
31 3	
32 *	
33 +	
34 END	
01♦LBL "DIS"	Function plotting routine
02 -5	
03 STO 00	Ymin
04 CHS	
05 STO 01	Ymax
06 133	
07 STO 02	Plot width
08 -4	
09 STO 08	Xmin
10 CHS	
11 STO 09	Xmax
12 .5	
13 STO 10	X increment
14 "X3"	
15 ASTO 15	Function name #1

```
16 "X-3"
17 ASTO 16          Function name #2
18 "X-3-"
19 ASTO 17          Function name #3
20 SF 06
21 CF 05            Disappearing mode
22 3                # functions plotted
23 XROM "MP"        Call to MP
24 END
```

```
| X3
: X-3
ı X-3-
Y: -5.000 TO 5.000
X: -4.000 TO 4.000
ΔX=0.500
------------
-4.0
-3.5
-3.0
-2.5 ı
-2.0     ı
-1.5   |   ı
-1.0  :    |
-0.5      :ıı
 0.0        |
 0.5        | :ı
 1.0          |   :
 1.5              |
 2.0
 2.5
 3.0
 3.5
 4.0
------------
```

Figure 15. Three functions of Example 13 plotted simultaneously using MP in 'disappearing' overflow mode. In this example, the behavior of the 3 functions is examined in the region of the origin. When the functions exceed the Y limits of -5 to +5, they are not plotted, and are therefore less confusing to the investigator. Execution time: 5 min 13 sec.

G.3. 'Wraparound' Overflow Mode.

When the points of a function which exceed specified Y limits are of importance and are therefore to be displayed, but mirror plotting is not acceptable, 'wraparound' plotting may be used. Function points which are greater than Ymax or smaller than Ymin will be plotted coming back from the opposite edge of the plot. The wrapped around section will thus have its actual shape, but it will be shifted by a value equivalent to the width of the plot itself (Ymax - Ymin). If the function value is so large as to exceed additional limits after being wrapped around once, it will be wrapped around repeatedly, in the same fashion as mirror plotting. For this mode, flag 05 is set and flag 06 is clear.

Example 14. Plot the functions Y=1.5 sin X and Y=4 sin X from 0 to 180 degrees in increments of 5 degrees. Let the Y limits be -1 & 1. Use 'wraparound' plotting mode (SF05, Use symbols 1 and 2.

APPLICATION PROGRAM FOR: MP

```
01♦LBL "1"          Function #1
02 STO 29           Save X in R29
03 RCL 06
04 STO 30           Save R06 in R30
05 2
06 STO 06           CP skip index
07 RCL 29
08 XEQ "CP"         Call to CP
09 RCL 30
10 STO 06           Restore R06 from R30
11 RCL 29           Restore X from R29
12 SIN
13 1.5
14 *
15 RTN
16♦LBL "4"          Function #2
17 SIN
18 4
19 *
20 END

01♦LBL "WR"         Function plotting
02 XROM "RF"        routine
03 SF 21
04 CF 29            Clear user flags,
05 55               SF21, CF29, SF55
06 XROM "IF"
07 -1
08 STO 00           Ymin
09 CHS
10 STO 01           Ymax
11 140
12 STO 02           Plot width
13 0
14 STO 08           Xmin
15 180
16 STO 09           Xmax
17 15
18 STO 10           X increment
19 "1"
20 ASTO 15          Function nema #1
21 "4"
22 ASTO 16          Function name #2
23 CF 06
24 SF 05            Wraparound mode
25 2                # functions plotted
26 XROM "MP"        Call to MP
27 END
```

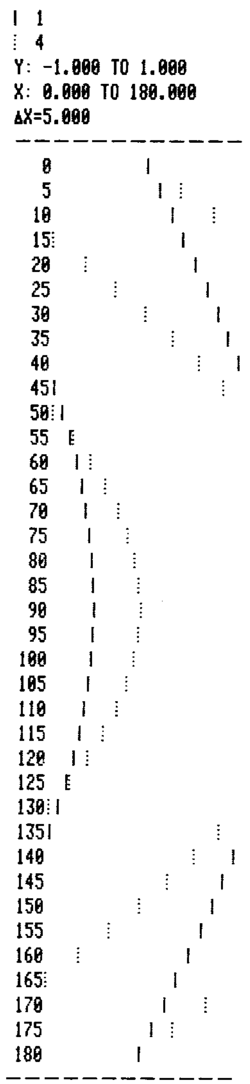


Figure 16. Two functions of Example 14 plotted using MP in the 'wraparound' overflow mode. Note in this example that the first function wraps around once, since it exceeds the positive Y limit by 50%. The 2nd function is wrapped around a second time, since its value at 90 degrees is 4, which is twice the (Y max-Y min) value. Had this function been larger, it would have wrapped around repeatedly until the value could be plotted. Execution time: 9 min 6 sec.

G.4. Mixed Overflow Modes.

Another way the various overflow modes can be used is to plot different functions using different overflow conditions. The states of flags F05 and F06 can be set within the functions themselves, so that the function values plotted for each are in the necessary position according to the overflow mode prescribed. This is illustrated in Example 15.

Example 15. Plot the 3 functions of example 13 using mirror plotting, wraparound plotting and clipped plotting in functions 1,2 and 3 respectively.

The plotting routine in example 13 would remain the same, except for removal of steps 20 and 21 which originally fixed the overflow to disappearing' mode.

APPLICATION PROGRAM FOR: MP	
01♦LBL "X3"	First function
02 FIX 1	
03 ACX	X labelling
04 1	
05 SKPCHR	
06 RDN	
07 3	
08 Y↑X	
09 FIX 0	
10 SF 05	
11 SF 06	Mirror plotting
12 RTN	
13♦LBL "X-3"	Function #2
14 ENTER↑	
15 3	
16 Y↑X	
17 X<>Y	
18 3	
19 *	
20 +	
21 SF 05	
22 CF 06	Wraparound plotting
23 RTN	
24♦LBL "X-3-"	Function #3
25 ENTER↑	
26 RCL Y	
27 3	
28 Y↑X	
29 X<>Y	
30 X↑2	
31 3	
32 *	
33 +	
34 X<>Y	
35 3	
36 *	
37 +	
38 CF 06	
39 CF 05	Clipped plotting
40 END	
01♦LBL "DIS"	Function plotting routine
02 -5	
03 STO 00	
04 CHS	
05 STO 01	
06 133	
07 STO 02	
08 -4	
09 STO 08	
10 CHS	
11 STO 09	
12 .5	
13 STO 10	
14 "X3"	
15 ASTO 15	
16 "X-3"	
17 ASTO 16	
18 "X-3-"	
19 ASTO 17	
20 3	No overflow mode settings here
21 XROM "MP"	
22 END	

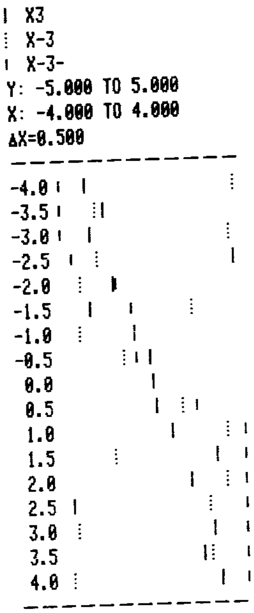
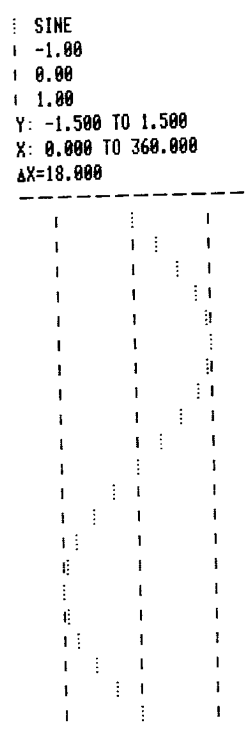


Figure 17. The 3 functions of Example 13 plotted for Example 15 with different overflow modes, each designated within the plotted function programs themselves. Execution time: 5 min 31 sec.

APPLICATION PROGRAM FOR: MP	
01♦LBL "SINE"	Function plotted
02 SIN	
03 RTN	
04♦LBL "SX"	Function plotting routine
05 XROM "RF"	
06 SF 21	Clear user flags, F21, SF55
07 55	
08 XROM "IF"	
09 -1.5	
10 STO 00	Ymin
11 CHS	
12 STO 01	Ymax
13 168	
14 STO 02	Plot width
15 0	
16 STO 08	Xmin
17 360	
18 STO 09	Xmax
19 18	
20 STO 10	X increment
21 .2333	
22 STO 12	Remap 1st 4 symbols
23 SF 04	
24 "SINE"	
25 ASTO 15	Function name
26 -1	
27 STO 16	Axis #1
28 0	
29 STO 17	Axis #2
30 1	
31 STO 18	Axis #3
32 4	# 'functions'
33 XROM "MP"	Call to MP
34 END	

H. X Axes in Plots.

Often when functions are plotted, it is convenient to have one or more 'axes' running along the length of the plot, at various Y heights. An example would be to have an X axis at Y=0 run down the center of a conventional sinusoidal function which ranges from Y=-1 to Y=1. Back in Example 10, the function Y=4 was plotted using a program LBL 'TH' 4 RTN, with the global label 'TH' stored in register 16 as the second function to be plotted. This is unnecessary for plotting constants such as Y=4. All that is needed is to store the constant to be plotted into the appropriate function name register, and MP takes care of the rest. Had the number 4 been stored in R16, it would be plotted as Y=4 using the symbol designated from R12. MP checks the contents of R15 through R23. If a constant is present, it is plotted as an axis; if ALPHA information is present, MP searches for the corresponding global label.



Example 16. Plot the Y=sin X curve from 0 to 360 degrees with 18 degree increments. Use symbol number 2 for the function itself. Also plot axes at Y=1, 0 and -1, using symbol #3 for each. Let the Y limits be -1.5 and 1.5.

Figure 18. Plot of Example 16 for a sine curve with three X-axes also plotted. Execution time: 6 min 57 seconds.

I. Prompting for User Inputs to MP.

Because of the large number of inputs to the MP routine, it may be inconvenient to remember where all the input information belongs. The following program provides some assistance by prompting the user for all the basic inputs to MP: function names, Ymin, Ymax, plot width, Xmin, Xmax and X increment. It then calls the MP routine. Simply set all the flags to their correct status, set the other options appropriately, and XEQ MPP. The listing is presented below:

APPLICATION PROGRAM FOR: MP	
01♦LBL "MPP"	
02 SF 08	
03 GTO 00	
04♦LBL "HPP"	
05 CF 08	
06♦LBL 00	
07 "NO. FCNS?"	Input # of functions
08 PROMPT	
09 STO 04	
10 1 E3	
11 /	
12 15.014	
13 +	
14 STO 03	
15 FIX 0	
16♦LBL 01	
17 "NAME "	Input each name or X axis value
18 RCL 03	
19 14	
20 -	
21 ARCL X	
22 "⊢?"	
23 AON	
24 PROMPT	
25 FS? 48	
26 ASTO IND 03	
27 FC? 48	
28 STO IND 03	
29 ISG 03	
30 GTO 01	
31 "Y MIN?"	Input Ymin
32 PROMPT	
33 STO 00	
34 "Y MAX?"	Input Ymax
35 PROMPT	
36 STO 01	
37 "PLOT WIDTH?"	Input plot width
38 PROMPT	
39 STO 02	
40 "X MIN?"	Input Xmin
41 PROMPT	
42 STO 08	
43 "X MAX?"	Input Xmax
44 PROMPT	
45 STO 09	
46 "X INC?"	Input X increment
47 PROMPT	
48 STO 10	
49 RCL 04	
50 FIX 4	
51 FS? 08	
52 XROM "MP"	Calls MP or HP
53 FC? 08	
54 XROM "HP"	
55 END	

This routine may also be used for passing inputs to HP by pressing XEQ HPP . In that case, HP would be executed as the final step. If estimated execution times are also desired, one could replace the lines XROM MP and XROM HP with XEQ MPT and XEQ HPT respectively. Then, after all prompting, the run time would be printed before the plot routine was executed.

The barcode for MPP/HPP appears in Appendix N.

J. Advanced MP Applications:

Many of the various features of the MP routine can be combined to perform plotting tasks which might be considered advanced applications to the novice PPC ROM user. Some of these are described in this section. They include:

1. Plotting up to 3 functions simultaneously using 3-column-wide plot symbols
2. Changing the plot symbol of the X axis every Nth plot line
3. X axes that include ´tic marks´ at every Nth plot line
4. Automatic computation of limits in the Y direction necessary to plot functions with unknown behavior
5. Generating plots which span multiple widths of printer paper (´superplots´)
6. Using MP to generate and sort multiple function plot data and then exiting to a user´s special RAM plotting program

A short discussion is provided for each of the above ideas to demonstrate the capabilities of the MP routine .

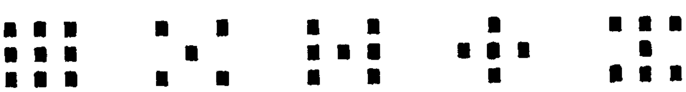
J.1. Triple Width Symbols.

Up to three functions may be plotted simultaneously by MP in its normal function-plotting mode with each function using plot symbols that are 3 columns wide. This requires plotting three points for each function (which corresponds to actually plotting 9 functions) and forcing 2 of the 3 points to plot their symbols exactly one column to the left and one column to the right of the central symbol. The value stored by MP into register R03 is very useful in this endeavor. It is the reciprocal of the distance (in real Y-directional units) between consecutive thermal print dots, based on the Y minimum and Y maximum that have been placed in R00 and R01 and on the plot width in R02. The format for creating a 3 column function symbol is as follows:

```
 01 LBL 'MAIN'   Main fcn. (center point)
  .     .
  .     .
  .     .
 nn last line
 nn+1 STO 35        Save in R35 before RTN
 nn+2 RTN
 nn+3 LBL 'LEFT'    Function for point on
 nn+4 RCL 35        left side
 nn+5 RCL 03
 nn+6  1/X          Subtract one column width
 nn+7   -           from R35 (center point)
 nn+8  RTN
 nn+9 LBL 'RIGHT'   Function for point  on
 nn+10 RCL 35       right
 nn+11 RCL 03
 nn+12  1/X         Add 1 column width to R35
 nn+13   +
 nn+14  END
```

Register 35 was chosen to temporarily save the output value from the function, so that it can be passed on as input to the other 2 programs, 'LEFT' and 'RIGHT'. R35 is the lowest available register when 9 functions are plotted along with the use of both user-defined plot symbols.

Now, place the 3 function names in three data registers, such as R15, 16 and 17. Make sure that the main function (which plots the center point) is the first of the three to be called by MP . This is necessary since register 35 must contain a meaningful value before the other two programs can be executed correctly. A good rule of thumb would be to store the main function name followed by the left-side function name, and then the right-side function name. The symbol mapping in register R12 should be chosen carefully so that the desired 3-column symbol is achieved. If it is 'built' from standard symbols 5,6, and/or 7, a number of variations are possible. Consider these examples:

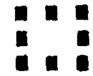


5 5 5   6 7 6   5 7 5   7 5 7   6 5 6

Figure 19. Five 3 column symbols constructed by merging 3 standard symbols side by side. The numbers of the symbols chosen are under the symbols themselves.

Symbol choices may be affected by the fact that the thermal dots appear to merge together in the direction across the printed line, but do not merge down along the length of the paper. Test out various symbol combinations to find a desirable looking choice.

The order in which the function names are stored in registers R15 and up will not correspond to the order of the actual appearance of the thermal dots in any 3-column symbol, since the central column must be specified first. Thus, a symbol which looks like this:

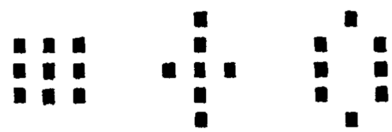


Symbol # 5 6 5

will be mapped as '.655' in R12, not '.565' as the symbol appears.

In a situation where the maximum of three 3-column symbols are plotted at the same time, registers R35, 36 and 37 are the first available places for temporary storage of the 3 main-function outputs for passing on to the side functions.

Example 17. Plot the 3 functions Y=sin X, Y=cos X and Y = -cos X simultaneously using 3 triple plot symbols. Use the following symbols for the functions:



Symbol #   5 5 5   7 3 7   5 8 5
(user symbol #8 is ACCOL #34)

Let the Y limits be -1.05 and 1.05 (to reveal the outer dots of the symbols when the central column values equal ± 1), X limits be 0 and 360 with an increment of 18 degrees.

Label the 3 main functions 'SINE', 'COSINE', and '-COS', with temporary storage of output values going into R35, 36 and 37, respectively. Call the side column functions 'SL' and 'SR', 'CL' and 'CR', and '-CL' and '-CR'. Store the symbol map .555377855 in R12 with F04 set and the value .034 in R33 for user symbol #8.

APPLICATION PROGRAM FOR:	MP
01♦LBL "SINE" 02 SIN 03 STO 35 04 RTN	1st main function
05♦LBL "SL" 06 RCL 35 07 RCL 03 08 1/X 09 - 10 RTN	1st left point
11♦LBL "SR" 12 RCL 35 13 RCL 03 14 1/X 15 + 16 RTN	1st right point
17♦LBL "COSINE" 18 COS 19 STO 36 20 RTN	2nd main function
21♦LBL "CL" 22 RCL 36 23 RCL 03 24 1/X 25 - 26 RTN	2nd left point

27♦LBL "CR"	2nd right point
28 RCL 36	
29 RCL 03	
30 1/X	
31 +	
32 RTN	
33♦LBL "-COS"	3rd main function
34 COS	
35 CHS	
36 STO 37	
37 RTN	
38♦LBL "-CL"	3rd left side
39 RCL 37	
40 RCL 03	
41 1/X	
42 -	
43 RTN	
44♦LBL "-CR"	3rd right side
45 RCL 37	
46 RCL 03	
47 1/X	
48 +	
49 END	
01♦LBL "3FCN"	Plotting routine
02 XROM "RF"	
03 SF 21	Clear user flags,
04 55	SF21, SF55
05 XROM "IF"	
06 -1.05	
07 STO 00	Ymin
08 CHS	
09 STO 01	Ymax
10 168	
11 STO 02	Plot width
12 0	
13 STO 08	Xmin
14 360	
15 STO 09	Xmax
16 18	
17 STO 10	X increment
18 .555377855	
19 STO 12	Remap plot symbols
20 SF 04	
21 .034	
22 STO 33	Set user symbol #8
23 "SINE"	
24 ASTO 15	
25 "SL"	
26 ASTO 16	
27 "SR"	
28 ASTO 17	
29 "COSINE"	
30 ASTO 18	Function names
31 "CL"	stored in R15-R23
32 ASTO 19	
33 "CR"	
34 ASTO 20	
35 "-COS"	
36 ASTO 21	
37 "-CL"	
38 ASTO 22	
39 "-CR"	
40 ASTO 23	
41 9	# functions plotted
42 XROM "MP"	Call to MP
43 END	

```
ı SINE
ı SL
ı SR
ı COSINE
· CL
· CR
: -COS
ı -CL
ı -CR
Y: -1.050 TO 1.050
X: 0.000 TO 360.000
ΔX=18.000
```

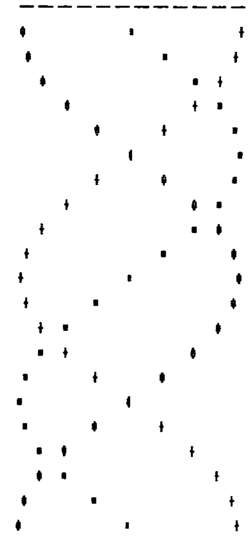


Figure 20. Three functions of Example 17 plotted simultaneously by MP using symbols that are each 3 columns wide. The symbols are plotted using 3 functions mapped to 3 consecutive columns in the printed line, with each main function mapped to the central column of its symbol. Execution time: 14 min 4 sec.

It can be seen in Figure 20 that the plot symbols for Y=0 are only 2 columns wide at X=0, 90, 180, 270, and 360. This results from the central and right column-number calculations rounding off to the same values for these particular Y limits and field width. This subject will be discussed further in Section J.6.

J.2. Plots With X Axis Symbol Changing Every Nth Line.

In addition to X-axis numeric labelling on a plot, it may be useful sometimes to have an X axis that changes to a different plot symbol for every Nth line that is printed. This would serve as an additional reminder of the X value at various points along the functions. To achieve this, the X axis is plotted by a function that includes special tests and then returns a constant, rather than by simply storing the numeric constant in the function name register. The X-axis function tests for the number of the printed line to determine whether the symbol for the axis should be one choice or the other. The symbol number for the current line is then stored directly into R12 before the X-axis program ends. The following program example was written by Jack Sutton (5622) to demonstrate this capability:

Example 18. Plot the 3 functions Y = sin 3X, Y= 0.5 sin X and Y= sin 3X + 0.5 sin X simultaneously. Let the Y limits be -1.5 and 1.5, let the X limits be 0 and 360 degrees with an increment of 10 degrees and include numeric labels in the X direction. Use symbols 4,2 and 5 for the 3 functions, respectively. Also position an X axis at Y=0 using symbol #1, which changes to symbol #7 every 5th printed line.

APPLICATION PROGRAM FOR:	MP
01♦LBL "DEMOAXS"	Plotting routine
02 .038	
03 XROM "BC"	Clear R00 - R38
04 CF 29	
05 SF 04	Symbol map override
06 1.5	
07 STO 01	Ymax
08 CHS	
09 STO 00	Ymin
10 140	
11 STO 02	Plot width
12 360	
13 STO 09	Xmax
14 10	
15 STO 10	X increment
16 .1425	
17 STO 12	1st symbol map
18 "AXIS"	
19 ASTO 15	
20 "SINE3"	
21 ASTO 16	Store function names
22 "SINE"	in R15 - R18
23 ASTO 17	
24 "SINET"	
25 ASTO 18	
26 4	
27 STO 35	Counter for 5th line
28 1	
29 +	
30 STO 37	Counter reset value
31 4	# functions
32 XROM "MP"	Call to MP
33 XROM "PO"	Advance paper out
34 RTN	
35♦LBL "SINE3"	Function #1
36 3	
37 *	
38 SIN	
39 STO 36	
40 RTN	
41♦LBL "SINE"	Function #2
42 SIN	
43 .5	
44 *	
45 ST+ 36	
46 RTN	
47♦LBL "SINET"	Function #3
48 RCL 36	
49 RTN	
50♦LBL "AXIS"	Axis function
51 RCL 06	
52 STO 38	Save R06 in R38
53 2	
54 STO 06	Set CP skip index
55 RCL 08	X label value
56 XROM "CP"	Call to CP
57 RCL 38	Restore R06
58 STO 06	
59 DSE 35	
60 SF 00	
61 FS? 00	
62 GTO 01	Test for 5th line,
63 RCL 37	switch symbol maps
64 STO 35	
65 .7425	
66♦LBL 01	
67 FS?C 00	
68 .1425	
69 STO 12	
70 0	Return axis value
71 .END.	

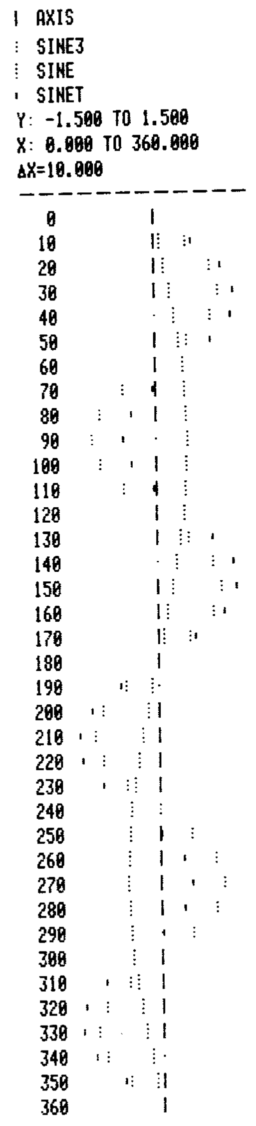


Figure 21. The 3 functions from Example 18 plotted simultaneously by MP, with the X axis character changed from symbol #1 to #7 every 5th line to mark every 50 degrees. Execution time: 14 min 13 seconds.

J.3. X Axes With Tic Marks:

Another way to identify periodic values along the X axis is to print 'tic marks' on the axis. This can be achieved in a manner similar to that for creating the 3-column-wide plot symbols shown in Example 17. The tic marks on the axis are formed by the addition of new functions which fall in thermal dot positions immediately adjacent to the axis column. These tic-functions must fall outside the Y limits of the entire plot for the intermediate X values, with the disappearing overflow mode used to avoid printing those lines.

Alternately, the values of the tic-mark functions could be set to zero for the intermediate X values. That way, they would be masked by the original X axis symbol using a higher ACCOL value.

Example 19. Plot the function Y = sin X + (1/3)sin 3X for X=0 to X=360 degrees in increments of 10 degrees. Let the Y limits be -1.5 and +1.5, with a plot width of 168 columns. Also, include an Xaxis at Y=0 with a 3-dot-wide tic mark to the right of the axis column every 30 degrees. Use symbol #2 for the function and #3 for the axis.

The tic mark should be in the middle dot of its column, which is standard symbol #7. It must fall beyond -1.5 or +1.5 (i.e., outside the plot field) in all lines except X=0,30, 60,90, etc. Since the tic mark is to be 3 columns wide, three separate functions will be used to print it, with each mapping one column farther to the right than the one before. Since the axis itself is a constant, no temporary storage is necessary for the tic-mark functions to use this value as an input, as was the case in 3-column symbol plotting.

APPLICATION PROGRAM FOR: MP

Program	Comment
01♦LBL "S3X"	Function plotted
02 SIN	
03 LASTX	
04 3	
05 *	
06 SIN	
07 3	
08 /	
09 +	
10 RTN	
11♦LBL "TIC1"	1st tic function
12 ISG 35	
13 GTO 00	
14 1.003	
15 STO 35	
16 RCL 03	
17 1/X	
18 RTN	
19♦LBL 00	
20 5	
21 RTN	
22♦LBL "TIC2"	2nd tic function
23 ISG 36	
24 GTO 01	
25 1.003	
26 STO 36	
27 RCL 03	
28 1/X	
29 2	
30 *	
31 RTN	
32♦LBL 01	
33 5	
34 RTN	
35♦LBL "TIC3"	3rd tic function
36 ISG 37	
37 GTO 02	
38 1.003	
39 STO 37	
40 RCL 03	
41 1/X	
42 3	
43 *	
44 RTN	
45♦LBL 02	
46 5	
47 RTN	
48♦LBL "SINX"	Plot routine
49 -1.5	
50 STO 00	Ymin
51 CHS	
52 STO 01	Ymax
53 168	
54 STO 02	Plot width
55 0	
56 STO 08	Xmin
57 360	
58 STO 09	Xmax
59 10	
60 STO 10	X increment
61 1.001	
62 STO 35	
63 STO 36	Set 3 tic counters
64 STO 37	
65 "S3X"	
66 ASTO 15	
67 "TIC1"	
68 ASTO 16	Store function names in R15 - R18
69 "TIC2"	
70 ASTO 17	
71 "TIC3"	
72 ASTO 18	
73 0	
74 STO 19	Axis at X=0
75 .27773	
76 STO 12	Remap symbols
77 SF 04	
78 SF 06	Disappearing mode
79 5	# functions
80 XROM "MP"	Call to MP
81 END	

```
S3X
TIC1
TIC2
TIC3
0.00
Y: -1.500 TO 1.500
X: 0.000 TO 360.000
ΔX=10.000
```

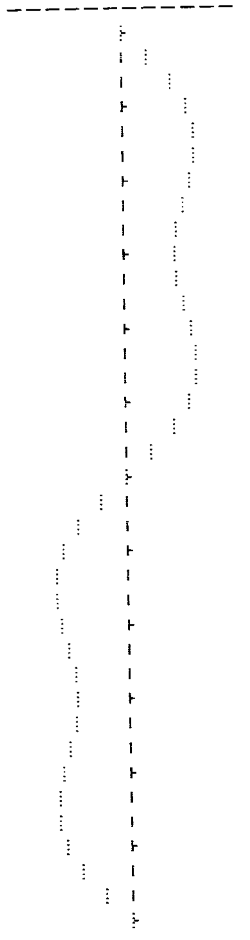


Figure 22. Plot of the function Y = sin X + (1/3)sin 3X from Example 19 with tic marks printed on the axis every 30 degrees. Execution time: 18 min 19 sec.

J.4. Automatic Computation of Y Limits of a Function.

In some cases, a user may wish to plot a function whose behavior is unknown in the desired range of X values. Here is a program written by Jack Sutton (5622) which accepts X inputs and will print the Y minimum and Y maximum of a given function for each 10 values of X between the X limits, inclusive. This program uses the ROM routine **BX** (Block Extremes) to find the Y limits for each 10-value range.

BAR CODE ON PAGE 479

APPLICATION PROGRAM FOR: **MP**

```
01♦LBL "MAXMIN"        39 ISG 00
02 CF 21               40 GTO "CAL"
03 .014                41♦LBL 02
04 XROM "BC"           42 "TO "
05 "X MIN?, R/S"       43 RCL 11
06 PROMPT              44 RCL 13
07 STO 11              45 -
08 "X MAX?, R/S"       46 ARCL X
09 PROMPT              47 "├ :"
10 STO 12              48 PRA
11 "X INC?, R/S"       49 RCL 00
12 PROMPT              50 1
13 STO 13              51 -
14 AON                 52 INT
15 "F(X) NAME"         53 1 E3
16 "├?, R/S"           54 /
17 PROMPT              55 1
18 ASTO 14             56 +
19 AOFF                57 XROM "BX"
20 SF 21               58 XEQ 00
21♦LBL "BLK"           59 RCL 12
22 "FROM "             60 RCL 11
23 ARCL 11             61 X<=Y?
24 PRA                 62 GTO "BLK"
25 1.01                63 ADV
26 STO 00              64 "DONE"
27 XROM "BC"           65 PRA
28♦LBL "CAL"           66 XROM "PO"
29 RCL 12              67 RTN
30 RCL 11              68♦LBL 00
31 X>Y?                69 "MIN="
32 GTO 02              70 ARCL X
33 XEQ IND 14          71 PRA
34 STO IND 00          72 "MAX="
35 RCL 11              73 ARCL Y
36 RCL 13              74 PRA
37 +                   75 ADV
38 STO 11              76 .END.
```

The barcode for MAXMIN appears in Appendix **N**.

Example 20. Use the MAXMIN program to find the Y limits for the function Y=3*X↑2+6*X+4 between the X values of -10 and 10, with X increment of X=0.2:

First, write the program for the function to be analyzed:

```
01♦LBL "POLY"
02 STO Y
03 X↑2
04 3
05 *
06 X<>Y
07 6
08 *
09 +
10 4
11 +
12 END
```

Now, XEQ MAXMIN and enter the data when prompted:

KEYSTROKES	DISPLAY	RESULT
XEQ MAXMIN	X MIN?, R/S	Prompt for Xmin
10 CHS R/S	X MAX?, R/S	Store Xmin, prompt
CHS R/S	X INC?, R/S	Store Xmax, prompt
.1 R/S	F(X) NAME?, R/S	Store Xinc, prompt
POLY R/S	----	Store Fcn name, then print Ymin,Ymax for each 10 X values

```
FROM -10.000        FROM 2.000
TO -8.200 :         TO 3.800 :
MIN=156.520         MIN=28.000
MAX=244.000         MAX=70.120

FROM -8.000         FROM 4.000
TO -6.200 :         TO 5.800 :
MIN=82.120          MIN=76.000
MAX=148.000         MAX=139.720

FROM -6.000         FROM 6.000
TO -4.200 :         TO 7.800 :
MIN=31.720          MIN=148.000
MAX=76.000          MAX=233.320

FROM -4.000         FROM 8.000
TO -2.200 :         TO 9.800 :
MIN=5.320           MIN=244.000
MAX=28.000          MAX=350.920

FROM -2.000         FROM 10.000
TO -0.200 :         TO 10.000 :
MIN=1.000           MIN=364.000
MAX=4.000           MAX=364.000

FROM 0.000
TO 1.800 :          DONE
MIN=4.000
MAX=24.520
```

Figure 23. Output of the MAXMIN program showing Y limits of the function in Example 20. These values may now be used to select the Ymin and Y max inputs to MP for plotting. Execution time: 3 min 16 sec

J.5. Plots Using Multiple Paper Widths - 'Superplotting'.

When higher plot resolution is desired in the Y direction (across the printer paper) than can be obtained with 168 columns, it is possible to plot graphs with MP which require multiple widths of printer paper. This has been referred to as 'superplotting'. The routine shown below takes care of the housekeeping involved in printing each section of the plot, re-initializes the inputs and increments the Y limits. The only difference between the inputs for this program and for MP is that Ymax is stored in R35 instead of R01, and a Y increment value (the desired width of each printed plot section) is stored in R36. After all the function names are stored, simply set the limits and XEQ SMP:

1. Place the function names in R15 and up
2. Set disappearing overflow mode (CF05,SF06) so functions jump from strip to strip
3. Store Xmin, Xmax and Xinc in R08,R09,& R10
4. Store plot width in R02
5. Store Ymin in R00, Ymax in <u>R35</u> and Yinc in <u>R36</u>
6. Enter the number of functions to be plotted
7. XEQ SMP, and the plot is printed, a strip at a time, moving from Ymin to Ymax in steps equal to the Y increment stored in R36.

The SMP listing is as follows:

APPLICATION PROGRAM FOR:	MP
01♦LBL "SMP"	MP superplotting
02 STO 38	Save # fcns in R38
03 RCL 08	
04 STO 37	Xmin in R37
05 RCL 00	
06 RCL 36	
07 +	
08 STO 01	Ymin + Yincrement
09♦LBL 00	
10 RCL 38	Restore # fcns
11 XROM "MP"	Call to MP
12 RCL 01	
13 RCL 35	
14 X<=Y?	If done, stop
15 RTN	
16 RDN	If not, increment Ymin, Ymax
17 STO 00	
18 RCL 36	
19 ST+ 01	
20 RCL 37	
21 STO 08	
22 GTO 00	
23♦LBL "SHP"	HP superplotting
24 STO 45	Save # fcns in R45
25 RCL 08	
26 STO 44	X min in R44
27 RCL 00	
28 RCL 43	
29 +	
30 STO 01	Ymin + Yincrement
31♦LBL 01	
32 RCL 45	Restore # fcns
33 XROM "HP"	Call to HP
34 RCL 01	
35 RCL 42	
36 X<=Y?	If done, stop
37 RTN	
38 RDN	If not, increment Ymin, Ymax
39 STO 00	
40 RCL 43	
41 ST+ 01	
42 RCL 44	
43 STO 08	
44 GTO 01	
45 END	

The first plot strip has Ymin=Ymin and Ymax= Ymin+Yinc. The next strip has Ymin= the previous Ymax and Ymax=(new Ymin)+Yinc. This process repeats until the current Ymax exceeds that which was stored into R35. If Yinc is not chosen properly, the last plot strip will exceed the designated upper limit in the Y direction, but the excess may be removed by the user with a scissors if so desired.

Note that the SMP program listing also includes SHP, which is the superplotting routine for high resolution plotting with HP. See the HP write-up on page 188 of this manual for a complete description of SHP. The barcode for SMP/SHP appears in Appendix N.

Example 21. Use MP superplotting to plot the following 2 functions: Y=X↑4+3*X↑3-35*X↑2+5*X-4 and Y=2*X↑3+5*X↑2-25*X-175 simultaneously. Use Y limits of -750 and 450 with a Y increment of 400 (3 strips wide). Let the X limits be -8 and +6, with an X increment of 0.25. Use symbols #1 and #2 for the 2 functions, and also plot X axes at Y=-500, Y=0, and Y=250 using symbol #4 for each.

APPLICATION PROGRAM FOR:	MP
01♦LBL "X4" 02 STO 39 03 4 04 Y↑X 05 RCL 39 06 3 07 Y↑X 08 3 09 * 10 + 11 RCL 39 12 X↑2 13 35 14 * 15 - 16 RCL 39 17 5 18 * 19 + 20 4 21 -	22 RTN 23♦LBL "XX3" 24 STO 39 25 3 26 Y↑X 27 2 28 * 29 RCL 39 30 X↑2 31 5 32 * 33 + 34 RCL 39 35 25 36 * 37 - 38 175 39 - 40 RTN 41 END

01♦LBL "S2"	Plot routine
02 XROM "RF"	Clear F00- F28,
03 SF 21	SF21, SF55
04 55	
05 XROM "IF"	
06 -8	
07 STO 08	Xmin
08 6	
09 STO 09	Xmax
10 .25	
11 STO 10	X increment
12 -750	
13 STO 00	Ymin
14 450	
15 STO 35	Ymax
16 400	
17 STO 36	Y increment
18 SF 06	Disappearing mode
19 "X4"	
20 ASTO 15	Store function names
21 "XX3"	
22 ASTO 16	
23 -500	
24 STO 17	
25 0	Store axis values
26 STO 18	
27 250	
28 STO 19	
29 .12444	
30 STO 12	Specify symbols
31 SF 04	
32 5	# functions
33 XEQ "SMP"	Call to SMP
34 END	

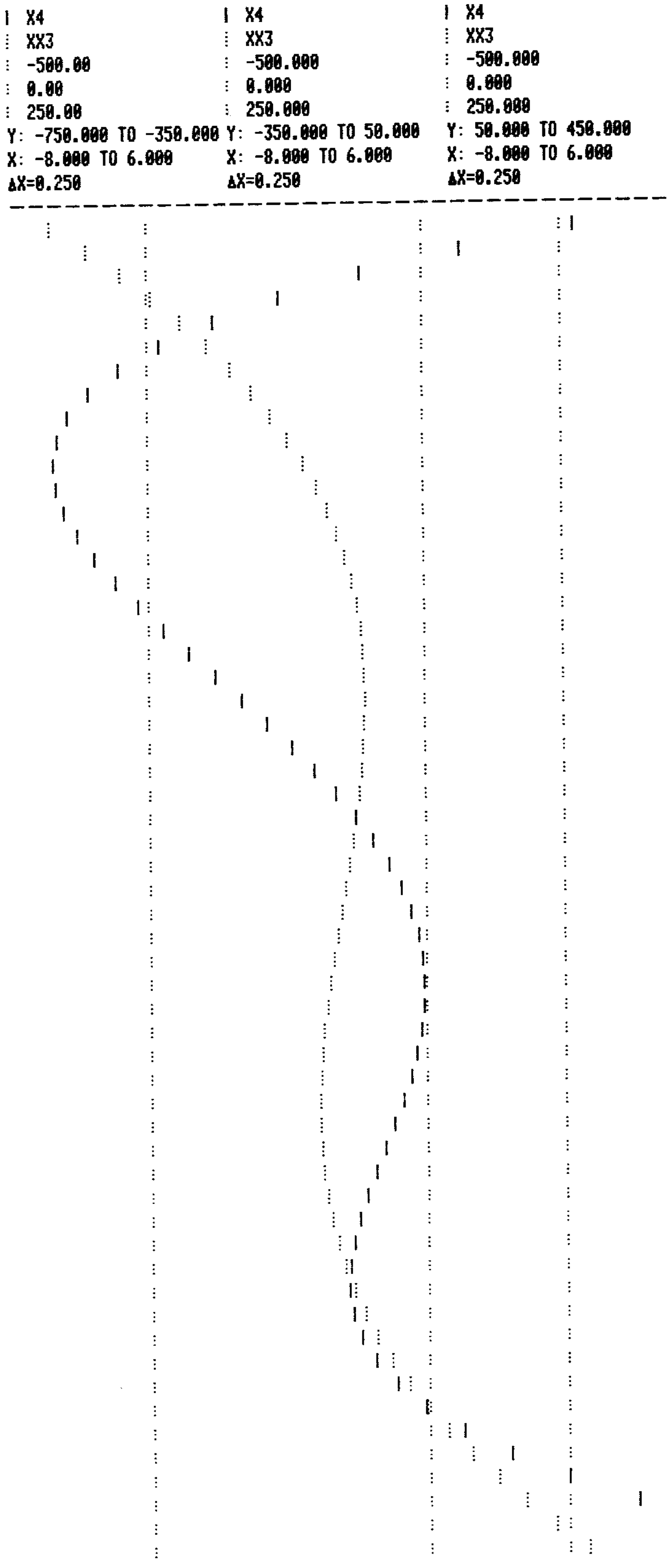


Figure 24. Superplot of the two functions from Example 21 made by MP under the control of SMP. The three strips were cut by hand and attached together edge to edge. All 3 strips used the full 168 column width for plotting. Execution time: 79 min 50 sec.

In doing multiple-strip plotting, one must be extremely careful when adding X-direction numeric labels to the strips. If any strip contains X labels, that strip cannot use the full width of the printer paper for the plot. In order for the scaling to remain consistant from strip to strip with the Y increment of each strip corresponding to its plot width, either of two approaches may be chosen:
1) The plot width may be left the same from strip to strip (being less than 168 columns) so that each Y increment is the same number of columns wide; or 2) the plot width of all strips without X labels (i.e., after the first) may be the full 168 columns wide and the change in plot width can be compensated for by changing the Y limits of the 168-column strips.

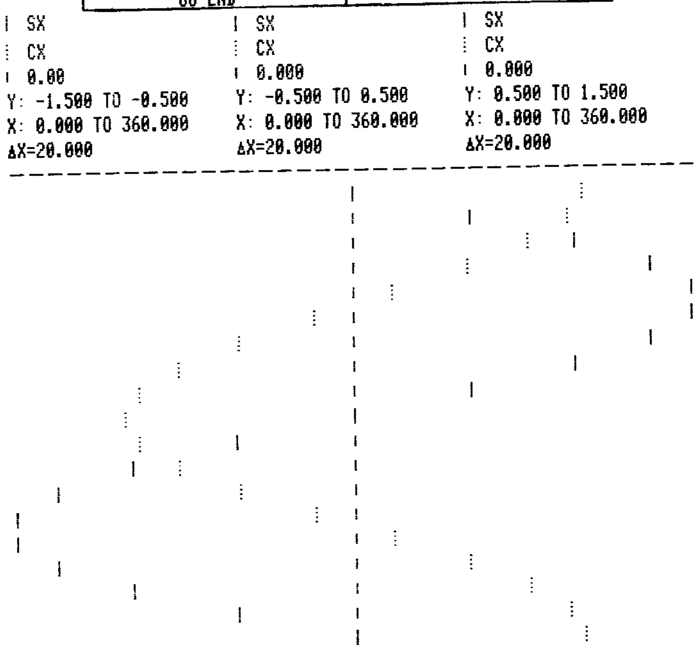
Example 22. Use MP superplotting to plot the functions Y= 1.5 sin X and Y=cos X simultaneously. Let the X limits be 0 and 360 degrees with increments of 20 degrees. Use Y limits of -1.5 and 1.5 with a Y increment of 1 (3 strips). Use symbols #1 and #2. Plot this graph twice: first, without X labelling using the full 168 columns per strip; and second, with X labelling in the first strip.

The first plot is straightforward and uses the SMP program with R02 set at 168 columns for each strip. The second plot requires calculation of the Y limits for each strip from the column widths, however, for proper scaling of the 3 strips. With CP used to accumulate the X labels into the print buffer and using 4 characters, 140 columns are left for the plot width of the first strip. The next two strips are 168 columns wide, which represent Y increments that are 168/140ths times the Y increment of the 140-column strip. The full plot width is 140+168+168 or 476 columns. The first Y increment is thus 140/176 of the full 3-strip width or 0.294*3=0.882, which gives Y limits of -1.5 to -0.618 for the first strip. The second and third strips are of equal widths with Y increments of 1.059 each, thus the Y limits are -0.618 to 0.441 and 0.441 to 1.5, respectively. Y-axis scaling is discussed further in Section J.6.

Because of the variations in the Y increment from plot strip to plot strip, the SMP program was not used for the second version of the superplot. Instead, special program SP was used to control the printing of each strip in sequence with the correct Y limits. Alternately, SMP could have been modified to test for completion of the first strip and then change to the Y increment required for the second and third strips.

APPLICATION PROGRAM FOR: MP	
01♦LBL "PL"	1st plot routine
02 "SX"	
03 ASTO 15	Function names
04 "CX"	
05 ASTO 16	
06 0	
07 STO 17	Axis at 0
08 XROM "RF"	
09 SF 21	Clear user flags,
10 55	SF21, SF55
11 XROM "IF"	
12 SF 06	
13 CF 05	Disappearing mode
14 0	
15 STO 08	Xmin
16 360	
17 STO 09	Xmax
18 20	
19 STO 10	X increment
20 168	
21 STO 02	Plot width
22 -1.5	
23 STO 00	Ymin
24 CHS	
25 STO 35	Ymax
26 1	
27 STO 36	Y increment
28 3	# functions
29 XEQ "SMP"	Call to SMP
30 RTN	
31♦LBL "SX"	Function #1
32 SIN	
33 1.5	
34 *	
35 RTN	
36♦LBL "CX"	Function #2
37 COS	
38 END	
01♦LBL "SP"	2nd plot routine
02 XROM "RF"	
03 SF 21	
04 CF 29	Clear user flags,
05 55	SF21, CF29, SF55
06 XROM "IF"	
07 SF 06	Disappearing mode
08 0	
09 STO 08	Xmin
10 360	
11 STO 09	Xmax
12 20	
13 STO 10	X increment
14 -1.5	
15 STO 00	Ymin
16 -.617647059	
17 STO 01	Ymax
18 140	
19 STO 02	Plot width
20 "SX"	
21 ASTO 15	
22 "CX"	
23 ASTO 16	Function names
24 0	
25 STO 17	Axis at 0
26 2	# of 1st strip fcns
27 XROM "MP"	1st call to MP
28 0	
29 STO 08	Xmin
30 RCL 01	
31 STO 00	Ymin = old Ymax
32 .441176471	
33 STO 01	New Ymax
34 168	
35 STO 02	Plot width
36 SF 00	Set to skip X labels
37 3	# of 2nd strip fcns
38 XROM "MP"	2nd call to MP
39 0	
40 STO 08	Xmin
41 RCL 01	
42 STO 00	Ymin = old Ymax
43 1.5	
44 STO 01	New Ymax

```
45 2                  # of 3rd strip fcns
46 XROM "MP"          3rd call to MP
47 RTN
48♦LBL "SX"           Function #1
49 FS? 00
50 GTO 00
51 STO 35
52 RCL 06
53 STO 36             X labelling
54 2                  using CP
55 STO 06
56 RCL 35
57 XROM "CP"
58 RCL 36
59 STO 06
60 RCL 35
61♦LBL 00
62 SIN
63 1.5
64 *
65 RTN
66♦LBL "CX"           Function #2
67 COS
68 END
```



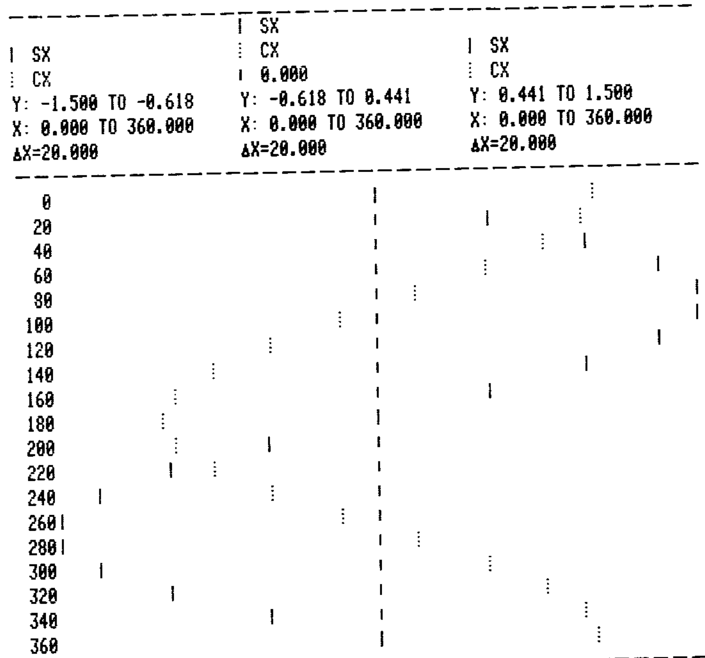


Figure 25. Plots of the functions Y=1.5sinX and Y=cosX using MP and two versions of superplotting discussed in Example 22. The upper plot used the SMP program and contains no X-numeric labels, while the lower plot did not use SMP but does include X-numeric labels. SMP was not used for the 2nd plot because the Y scaling changed from the first to the second strip, due to the increase in strip plot width. Execution time for the upper plot was 14 min 43 sec, and for the lower plot was 11 min 43 sec.

Another way to add X-direction numeric labels to an MP superplot is to simply print a row of X labels separately, and then paste it down along side the plot like all the other strips. This avoids the reduction in plot width that results with one labelled strip. In addition, custom headers, Y axes and Y numeric labelling are all possible with MP superplotting, just as in regular MP plotting. The following example includes a custom Y axis, both X and Y numeric labelling, an X axis, 2 user-defined plot symbols and a remapped plot-symbol order.

Example 23. To demonstrate that a square wave is an infinite sum of sine waves with decreasing amplitude and only odd harmonics, plot the functions Y=(1/I)*sin(I*X) for I=1, 3,5,7,9,11, and 13 simultaneously. Also plot 2 times the sum of all 7 functions. Let the X limits be 0 and 360 degrees with an increment of 5 degrees. Use SMP superplotting to achieve Y limits of -2 and +2 with a Y increment of 0.8 per plot strip (totalling 5 strips wide). Let each strip be 168 columns wide. Print a separate strip of X-direction numeric labels at intervals of 10 degrees (i.e., every other value of X), skip the standard header on each strip, and print a custom Y axis with numeric labelling at intervals of 0.5. Use symbol #1 for the sum of all 7 functions, symbols #2,3,4,5,6,7 and 8 for the 7 individual sine functions, and symbol #9 for an axis at 0. Let user-defined symbol #8 be ACCOL #65 and symbol #9 be ACCOL #99.

APPLICATION PROGRAM FOR: **MP**

```
01♦LBL "Y-AX"        Special Y axis rout.
02 GTO IND 40        Branch to nth section
03♦LBL 01            First section
04 FS? 00            Skip labels if FS00
05 GTO 11
06 "-2
07 "├ -1.5"
08 PRA               Numeric labels
09♦LBL 11
10 127
11 ACCOL
12 6
13 SKPCOL
14 "-----------"
15 ACA
16 "---"
17 ACA
18 127
19 ACCOL
20 6
21 SKPCOL
```

```
22 "---------"
23 ACA
24 PRBUF                1st axis drawn
25 FS? 00
26 ISG 40               Increment counter
27 0
28 XROM "IF"            Toggle F00
29 RTN
30♦LBL 02               2nd section
31 FS? 00
32 GTO 12               Skip labels if FS00
33 "     -1     "
34 "⊦       -.5"
35 PRA                  Numeric labels
36♦LBL 12
37 "------"
38 ACA
39 127
40 ACCOL
41 6
42 SKPCOL
43 "--------------"
44 ACA
45 127
46 ACCOL
47 6
48 SKPCOL
49 "--"
50 ACA
51 PRBUF                2nd axis drawn
52 FS? 00
53 ISG 40               Increment counter
54 0
55 XROM "IF"            Toggle F00
56 RTN
57♦LBL 03               3rd section
58 FS? 00
59 GTO 13               Skip labels if FS00
60 "          0."
61 "⊦0"
62 PRA                  Numeric labels
63♦LBL 13
64 "------------"
65 ACA
66 127
67 ACCOL
68 6
69 SKPCOL
70 "-----------"
71 ACA
72 PRBUF                3rd axis drawn
73 FS? 00
74 ISG 40               Increment counter
75 0
76 XROM "IF"            Toggle F00
77 RTN
78♦LBL 04               4th section
79 FS? 00
80 GTO 14               Skip labels if FS00
81 " 0.5       "
82 "⊦    1.0"            Numeric labels
83 PRA
84♦LBL 14
85 "---"
86 ACA
87 127
88 ACCOL
89 6
90 SKPCOL
91 "---------------"
92 ACA
93 127
94 ACCOL
95 6
96 SKPCOL
97 "-----"
```

```
98 ACA
99 PRBUF                4th axis drawn
100 FS? 00
101 ISG 40              Increment counter
102 0
103 XROM "IF"           Toggle F00
104 RTN
105♦LBL 05              5th section
106 FS? 00
107 GTO 15              Skip labels if FS00
108 "         1.5 "
109 "⊦            2"
110 PRA                 Numeric labels
111♦LBL 15
112 "---------"
113 ACA
114 127
115 ACCOL
116 6
117 SKPCOL
118 "-------------"
119 ACA
120 6
121 SKPCOL
122 127
123 ACCOL
124 PRBUF               5th axis drawn
125 SF 00
126 END
```

```
01♦LBL "ONE"            First function
02 SIN
03 STO 39
04 RTN
05♦LBL "TWO"            Second function
06 3
07 *
08 SIN
09 3
10 /
11 ST+ 39
12 RTN
13♦LBL "THR"            Third function
14 5
15 *
16 SIN
17 5
18 /
19 ST+ 39
20 RTN
21♦LBL "FOU"            4th function
22 7
23 *
24 SIN
25 7
26 /
27 ST+ 39
28 RTN
29♦LBL "FIV"            5th function
30 9
31 *
32 SIN
33 9
34 /
35 ST+ 39
36 RTN
37♦LBL "SIX"            6th function
38 11
39 *
40 SIN
41 11
42 /
43 ST+ 39
44 RTN
45♦LBL "SEV"            7th function
```

46 13	
47 *	
48 SIN	
49 13	
50 /	
51 ST+ 39	
52 RTN	
53♦LBL "NIN"	Last function (sum)
54 RCL 39	
55 2	
56 *	
57 RTN	
58♦LBL "PL9"	Plot routine
59 .3601	↑
60 STO 00	
61 2	
62 STO 06	
63 FIX 0	
64♦LBL 10	X labelling routine
65 RCL 00	
66 INT	
67 XROM "CP"	
68 PRBUF	
69 ADV	
70 ISG 00	
71 GTO 10	↓
72 0	
73 STO 08	Xmin
74 360	
75 STO 09	Xmax
76 5	
77 STO 10	X increment
78 -2	
79 STO 00	Ymin
80 CHS	
81 STO 35	Ymax
82 .8	
83 STO 36	Y increment
84 XROM "RF"	
85 SF 06	Disappearing mode
86 SF 21	Clear user flags,
87 55	SF21, SF55
88 XROM "IF"	
89 168	
90 STO 02	Plot width
91 "ONE"	
92 ASTO 15	Store function names
93 "TWO"	
94 ASTO 16	
95 "THR"	
96 ASTO 17	
97 "FOU"	
98 ASTO 18	
99 "FIV"	
100 ASTO 19	
101 "SIX"	
102 ASTO 20	
103 "SEV"	
104 ASTO 21	
105 0	
106 STO 22	Axis at 0
107 "NIN"	
108 ASTO 23	
109 .099	
110 STO 34	User symbol #9
111 .065	
112 STO 33	User symbol #8
113 .234567891	
114 STO 12	Symbol map
115 SF 04	
116 SF 07	Skip standard header
117 SF 09	
118 "Y-AX"	Special Y axis
119 ASTO 04	
120 1.005	
121 STO 40	Set counter R40
122 9	# functions
123 XEQ "SMP"	Call to SMP
124 END	

Figure 26 (Next page). SMP superplot of the 8 functions in Example 23. A special Y-axis printing routine was used to print a different axis and different numeric labelling with each of the five plot strips. X-axis labelling was done on a separate strip. Execution time: 5 hrs 11 min 19 sec.

J.6. Custom User Plot Routine and Scaled Y Axis.

A great deal of versatility has been demonstrated thus far for the MP routine. Even so, occasions may arise where the user will want special linear Y-axis scales and custom plot symbols other than the single-column symbols normally used with MP. Charlie Allen (4691) made detailed diagnostic runs of MP during its evaluation and determined ways to obtain these particular features. The programs and examples in this section illustrate the results of his efforts.

Two related sets of functions are used in the examples that follow in order to help compare plots made by the normal MP and the custom user plot routines. Linear Y-axis scaling is presented first to develop a better understanding of plotting relationships. Some instructive and useful MP diagnostic programs are then discussed. A custom user plot routine which uses MP line by line to generate and sort the function values to be plotted without printing the standard output concludes the section.

J.6.1. Custom Scaled Y Axis.

The ability of MP to provide Y-axis labels was shown in Section D, and these were used in several examples. For the Y-axis labels to accurately represent the plotted values, the Y-axis scale factor (stored in R03 by MP) must be correctly related to the Y-axis scale and labels. The purpose in this section is to explore that relationship in greater detail for use in the design and production of accurate linear Y-axis scales. This information is valuable for producing custom user plots also and will be used in Section J.6.3 for that purpose.

To better understand the Y-axis scale, consider a measuring instrument such as a 12" ruler. It begins at 0 and ends at 12, but it actually has 13 'tick' marks that could be numbered 1 to 13. The same is true of the Y-column scale; that is, a plot field width of 140 columns consists of the Y-column numbers 1 to 140, but the Y-column scale values are 0 to 139. Similarly, if 21 characters are used for a linear Y-axis scale, the Y-character numbers are 1 to 21 but the Y-character scale values are 0 to 20. This goes a step further than mere labelling and relates the Y-axis characters to the scale values.

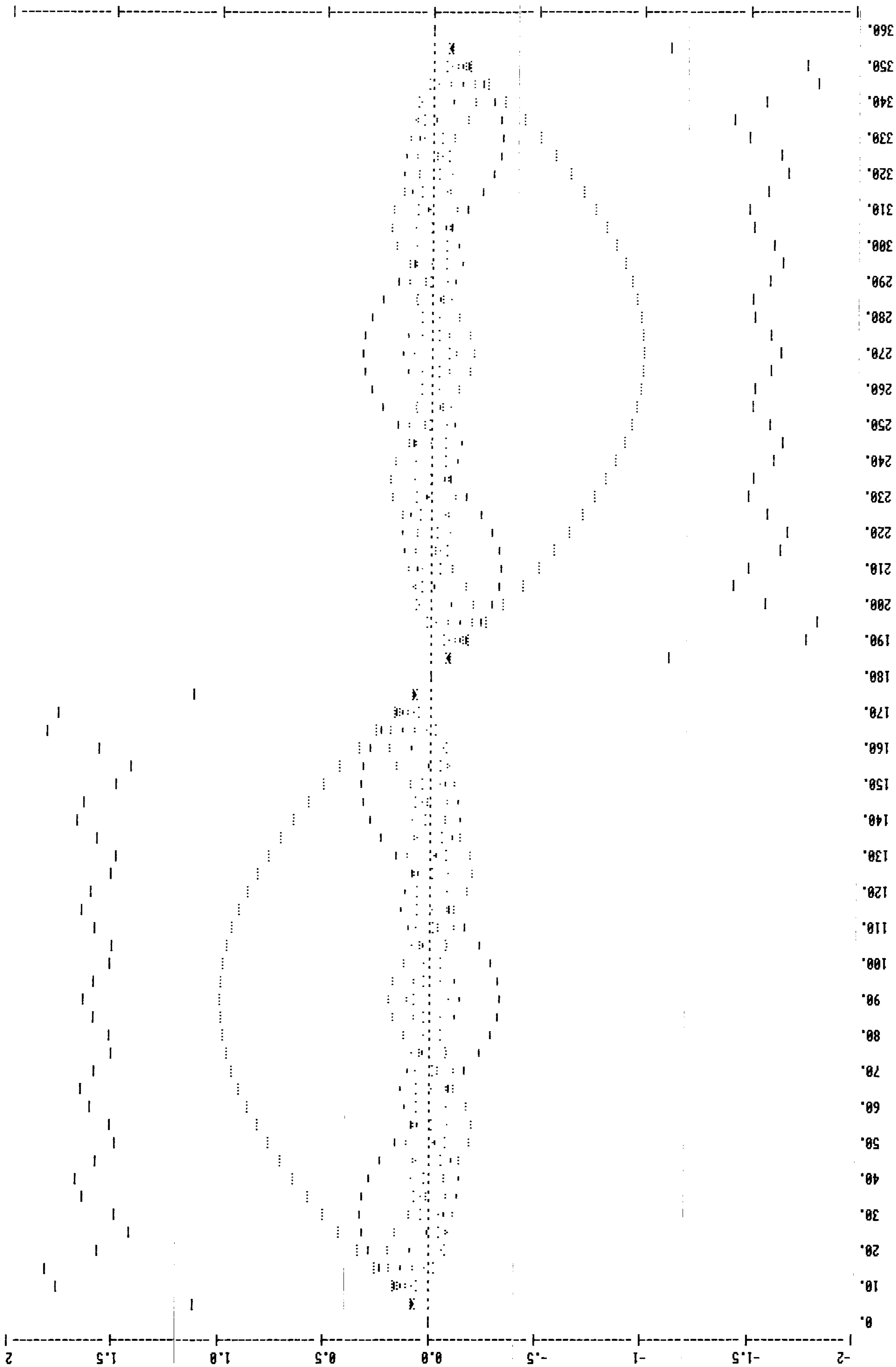


FIGURE 26.

A linear Y-axis scale has 7 times as many columns as characters, since there are seven columns per character. Some unique scales can be produced for special purposes if this is kept in mind while a Y-axis scale is being designed. In many cases, the plot range and behavior of a function are well known to the user. In those cases, custom tailor-made routines can be written for such features as X labels, custom Y axis with a linear scale and labels, and even a custom header or title block. The examples given in this section illustrate such special applications of MP..

Example 24. Plot a set of four sin X/X functions which represent the antenna patterns for 4 uniformly illuminated rectangular apertures of different sizes. Let the first function be the reference pattern (sin X/X) and the other 3 pattern functions be for apertures that are 1.5, 2/3 and 0.5 times the size of the first. Plot the functions in decibels, and use a custom linear Y-axis scale from 0 to -40 dB. Let the X limits be 0 to 12 radians with an X increment of 0.5.

With these X limits, the X-axis labels require 5 spaces or 35 columns, which leaves only 133 columns for the plot field width. The desired Y limits do not fit this plot width conveniently for a custom linear Y-axis scale, since the 133 columns corresponds to only 19 characters. A greater width can be used for the Y-axis scale, however, by letting it overlap into the X-label space. By using 21 characters for the scale (147 columns), the desired 0 to -40 dB range in Y corresponds to Y-character scale values of 0 to 20 measured from the center of the 1st to the center of the last character. The scale factor is then 2 dB per character, or 3.5 columns per dB.

With this scale, the corresponding field width of the full Y-axis scale is 141 columns, of which the first 8 columns are not part of the available plot field width. The actual plot field of 133 columns thus starts at -36.857 dB and ends at +0.857 dB, since the plot field extends 3 columns beyond the center of a scale character at each end rather than 3.5 columns, and 3/3.5 = 0.857 dB.

The program listing for the 4 functions is shown below. Each function calls the program 'APU', which is the basic sin X/X function in either dB with flag 00 clear or relative field with flag 00 set. The program 'AP1' also calls the correct X-axis label routine for each option of 'APU'. These same 4 functions are used in all the other examples in this section.

The custom Y-axis program 'AX1' and its companion X-axis label routine 'XL1' are also shown below. When X=0 at the start of a plot, 'AX1' prints a short custom header, the scale units, the Y-axis scale labels, and the custom linear Y-axis scale. When the plot is completed, the Y-axis scale is printed again. The plots for these examples were printed with flag F07 left cleared so as to print the standard header for information, and as ADV instruction was included in 'AX1' to separate the two.

Note that the custom Y-axis and X-axis label routines are paired up in this example. This is an important point, since the correct Ymin and Ymax limits to be stored in R00 and R01 depend on these two routines. For 'XL1', the plot field started immediately after the X-axis label, but this is not necessary as will be seen later in Example 25.

Two plots were made for Example 24, one without an X axis and the second one with an X axis at the -20 dB level. These are shown in Figures 27 (a) and (b). Inputs were entered manually and then printed by the 'INP' routine listed below. Line 09 of 'INP' was set to FIX 4 or 5 and line 13 was set to 15.018 or 15.019, depending on whether 4 or 5 functions were used (i.e., without or with an X axis), respectively.

APPLICATION PROGRAM FOR: **MP**	
01♦LBL "AP1" 02 FC? 00 03 XEQ "XL1" 04 FS? 00 05 XEQ "XL2" 06 XEQ "APU" 07 RTN	1st function X label for dB plots X label for Field plots
08♦LBL "AP2" 09 1.5 10 * 11 XEQ "APU" 12 RTN	2nd function
13♦LBL "AP3" 14 2 15 * 16 3 17 / 18 XEQ "APU" 19 RTN	3rd function
20♦LBL "AP4" 21 2 22 / 23 XEQ "APU" 24 END	4th function
01♦LBL "APU" 02 ENTER↑ 03 X=0? 04 GTO 00 05 R-D 06 SIN 07 X<>Y 08 / 09 GTO 01	Main Function (sin X/X)
10♦LBL 00 11 1	
12♦LBL 01 13 FC? 00 14 XEQ 02 15 RTN	Field if set, dB if clear
16♦LBL 02 17 ABS 18 LOG 19 20 20 * 21 END	

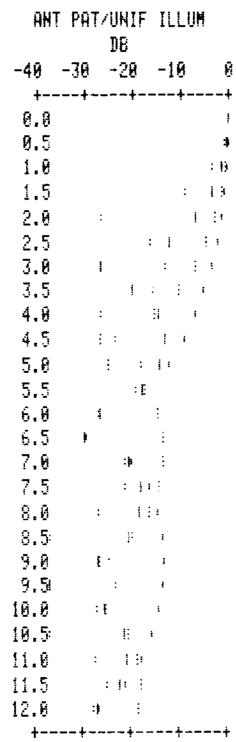
Program	Comment
01♦LBL "AX1"	Custom user Y axis routine for dB plots
02 RCL 08	
03 X=0?	
04 GTO 00	
05 GTO 01	
06♦LBL 00	
07 ADV	
08 " ANT PAT/"	Custom header
09 "⊦UNIF ILLUM"	
10 ACA	
11 PRBUF	
12 " "	Scale units
13 "⊦ DB"	
14 ACA	
15 PRBUF	
16 " -40"	Custom user Y-axis scale
17 "⊦ -30 -20"	
18 "⊦ -10 0"	
19 ACA	
20 PRBUF	
21♦LBL 01	Custom user Y axis
22 " +"	
23 "⊦----+----+"	
24 "⊦----+----+"	
25 ACA	
26 PRBUF	
27 RTN	
28♦LBL "XL1"	X-axis label routine for dB plots
29 10	
30 X<=Y?	
31 GTO 01	
32 1	
33 SKPCHR	
34 RDN	
35♦LBL 01	
36 RDN	
37 FIX 1	
38 RND	
39 ACX	
40 LASTX	Restores all decimal places in X
41 FIX 0	
42 END	
01♦LBL "INP"	Input print routine for 4 functions
02 "INPUT:"	
03 PRA	
04 FIX 3	
05 0.004	
06 PRREGX	
07 8.010	
08 PRREGX	
09 FIX 4	(FIX 5 for 5 functions)
10 12	
11 PRREGX	
12 FIX 3	
13 15.018	(15.019 for 5 functions)
14 PRREGX	
15 END	

Figure 27(a). Plot of the 4 functions of Example 24 in dB with no X axis. The plot has a custom user header, scale units, linear Y-axis scale, and custom linear Y axes. Execution time: 11 min 50 sec.

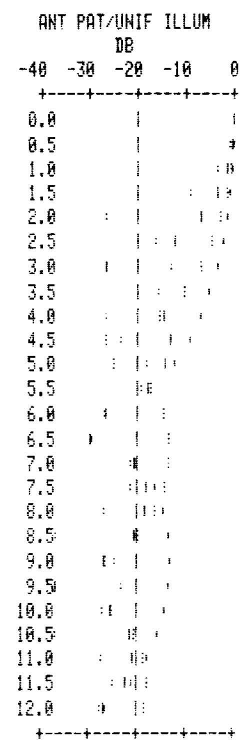
Figure 27(b). Plot of the 4 functions of Example 24 in dB with an X axis at -20 dB. The plot has a custom user header, scale units, linear Y-axis scale, and custom linear Y axes. Execution time: 14 min 10 sec.

(a)		(b)
PLOT OF AP1, 2, 3, 4 WITH AX1 AND XL1 AND CORRECT Y MIN, Y MAX FOR TRUE SCALE FACTOR		PLOT OF AP1, 2, 3, 4 WITH AX1 AND XL1 AND CORRECT Y MIN, Y MAX FOR TRUE SCALE FACTOR
INPUT:		INPUT:
R00= -36.857	Ymin	R00= -36.857
R01= 0.857	Ymax	R01= 0.857
R02= 133.000	Plot field width	R02= 133.000
R03= 3.500	Desired scale factor	R03= 3.500
R04= "AX1"	Custom Y-axis routine	R04= "AX1"
R08= 0.000	Xmin	R08= 0.000
R09= 12.000	Xmax	R09= 12.000
R10= 0.500	Delta X	R10= 0.500
R12= 0.3645	Symbol order	R12= 0.36451
R15= "AP1"	1st function	R15= "AP1"
R16= "AP2"	2nd function	R16= "AP2"
R17= "AP3"	3rd function	R17= "AP3"
R18= "AP4"	4th function	R18= "AP4"
		R19= -20.000
CF 00	Selects dB plot	CF 00
SF 04	Symbol order	SF 04
SF 09	Custom Y axis	SF 09
4.000 ENTER↑	# functions	5.000 ENTER↑

(a)	(b)
XROM "MP"	XROM "MP"
ı AP1	ı AP1
: AP2	: AP2
: AP3	: AP3
' AP4	' AP4
	ǀ -20.000
Y: -36.857 TO 0.857	Y: -36.857 TO 0.857
X: 0.000 TO 12.000	X: 0.000 TO 12.000
ΔX=0.500	ΔX=0.500



(a)



(b)

Example 25. Plot the set of 4 functions used in Example 24, but with flag F00 set to obtain a relative field plot. Use a custom linear Y-axis scale from -0.5 to 1.0, and plot an X-axis at Y=0. Let the X limits be 0 to 12 radians with an X increment of 0.5.

As before, the X-axis labels require 5 spaces or 35 columns, which leaves up to 133 columns for the plot field width. A different method is used in this example, however, to relate the custom linear Y-axis scale, the x-axis label space, and the plot field width. The desired range for the Y-axis scale suggests the use of 16 characters (112 columns), which gives Y-character scale values of 0 to 15 measured from the center of the 1st to the center of the last character. The scale factor is thus 0.1 per character, or 70 columns per unit Y value.

A plot field width of 106 columns is used, which just covers the desired range in Y inclusive (i.e., from center of the 1st character to center of the last). This is adequate with single-column plot symbols, since the 3-column margin at each end of the Y-axis scale is not needed. It will be seen in Section J.6.3, however, that some margin is needed for custom plotting with wider symbols in which case the method used in Example 24 is preferable.

The custom Y-axis program 'AX2' used for this plot and its companion X-axis label routine 'XL2' are shown below. 'AX2' is very similar to 'AX1' used in Example 24, except that the units, scale values, and the actual Y axis differ. 'XL2' has a significant difference from 'XL1', however, which is the inclusion of the steps 10, SKPCOL, RDN. These steps space the plot field from the X-label space by 1 character plus the 3-column margin at the left of the plot. The beauty of this method of designing the custom linear Y-axis scale is that the values of the Ymin and Ymax limits stored in R00 and R01 are actually equal to the scale label values at the ends of the custom Y-axis scale.

The plot for Example 25 is shown in Figure 28. As before, the inputs were entered manually and then printed by the 'INP' routine.

APPLICATION PROGRAM FOR: **MP**	
01♦LBL "AX2" 02 RCL 08 03 X=0? 04 GTO 00 05 GTO 01	Custom user Y axis routine for field plots
06♦LBL 00 07 ADV 08 " ANT PAT/" 09 "⊢UNIF ILLUM" 10 ACA 11 PRBUF	Custom header
12 " " 13 "⊢REL FLD" 14 ACA 15 PRBUF	Scale units
16 " " 17 "⊢ -0.5 0.0" 18 "⊢ 0.5 1.0"	Custom user Y-axis scale
19 ACA 20 PRBUF	
21♦LBL 01 22 " " 23 "⊢ +----+" 24 "⊢----+----+" 25 ACA 26 PRBUF 27 RTN	Custom user Y axis
28♦LBL "XL2" 29 10 30 X<=Y? 31 GTO 01 32 1 33 SKPCHR 34 RDN	X-axis label routine for field plots
35♦LBL 01 36 RDN 37 FIX 1 38 RND 39 ACX 40 LASTX 41 10	Restores all decimal places in X
42 SKPCOL 43 RDN 44 FIX 0 45 END	Separates plot field from X-label space

```
PLOT OF AP1, 2, 3, 4
WITH AX2 AND XL2
AND CORRECT Y MIN, Y MAX
FOR TRUE SCALE FACTOR

INPUT:

R00= -0.500    Ymin
R01=  1.000    Ymax
R02=  106.000  Plot field width
R03=  70.000   Desired scale factor
R04= "AX2"     Custom Y-axis routine

R08=  0.000    Xmin
R09=  12.000   Xmax
R10=  0.500    Delta X

R12=  0.36451  Symbol order

R15= "AP1"     1st function
R16= "AP2"     2nd function
R17= "AP3"     3rd function
R18= "AP4"     4th function
R19=  0.000    X axis

           SF 00   field plot
           SF 04   Symbol order
           SF 09   Custom Y axis
  5.000 ENTER↑     # functions
```

```
              XROM "MP"
     | AP1
     : AP2
     : AP3
     ' AP4
     | 0.000
     Y: -0.500 TO 1.000
     X: 0.000 TO 12.000
     ΔX=0.500
```

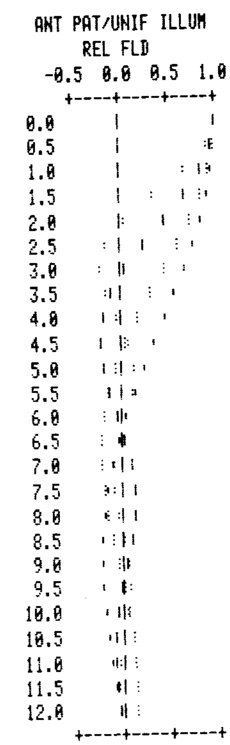


Figure 28. Plot of the 4 functions of Example 25 in Relative Field with an X axis at x=0. The plot has a custom user header, scale units, linear Y-axis scale, and custom linear Y axes.

J.6.2. MP Diagnostic Routine.

Two diagnostic routines that are useful for analyzing MP plot data are given in this section. The purpose is to help understand the contents of the MP output data for use in custom plotting. MP generates the SKPCOL and ACCOL plot data for each function in LBL 10 before printing a row. Originally, it was hoped that the four lines (7 bytes) FS?03, XEQ IND 35, FS?03, RTN could be inserted immediately after line 298 LBL 10 in MP (where the name of a custom user plot routine would be stored in R35 for custom plotting), but these could not be added to the ROM disk very easily so they were not included. Another method of accomplishing the desired result was therefore developed.

From diagnostic tests, it was found that the plot data required for the last row (the last value of X) remains in the MP print buffer registers R24 through R32 after a plot is finished. By clearing flag F21 before LBL 10 is executed, the plot data is generated, sorted, and made available in these registers without being printed. The plot diagnostic routines 'AXD' and 'AXC' that follow illustrate these features of MP.

Example 26. Plot the set of 4 functions used in Example 24 one line at a time for diagnostic purposes. Make the plot in dB with the same custom linear Y-axis scale that was used in Example 24. Plot the standard header and custom Y-axis scale with the first line, and only plot the pair of custom Y axes with subsequent lines. Print the plot data in the MP print buffer registers after each line.

The 'AXD' routine shown below is particularly useful for examining the behavior of the functions to be plotted by MP prior to performing custom plotting. By setting Xmax = Xmin, MP prints only a single line for one selected value of X. It then exits with the plot data available to be printed out by either PRREGX or the ROM routine BV. The former is used here because it is much faster for contiguous registers. 'AXD' ends in a short sequence of steps under LBL 03 that reset the inputs needed to execute MP again for the next incremental value of X.

The X-axis labels used with 'AXD' are produced by the same 'XL1' routine that was used in Example 24.

The diagnostic plots for Example 26 are shown in Figure 29 for 4 selected values of X. The first two values of X were specified by the inputs for Xmin, Xmax, and X increment in R08, R09 and R10. Executing MP again would have produced a plot for X=2. Instead, new values of Xmin, Xmax, and X increment were entered which produced plots for X=3 and X=6. It is seen from these plots that the set of plot data for X=3 is the only set that has values spaced sufficiently far apart for simplified custom plotting. The other 3 sets all require appropriate symbol overlap tests for use in custom plotting.

APPLICATION PROGRAM FOR:	MP
01♦LBL "AXD"	Custom user Y axis diagnostic routine for dB plots
02 RCL 08	
03 X=0?	
04 XEQ 00	Alternative form to GTO's of 'XL1'
05 XEQ 01	
06 XEQ 02	
07 RTN	
08♦LBL 00	
09 ADV	
10 " ANT PAT/"	Custom header
11 "⊢UNIF ILLUM"	
12 ACA	
13 PRBUF	
14 " "	Scale units
15 "⊢ DB"	
16 ACA	
17 PRBUF	
18 " -40"	Custom user Y-axis scale
19 "⊢ -30 -20"	
20 "⊢ -10 0"	
21 ACA	
22 PRBUF	
23 RTN	
24♦LBL 01	
25 " +"	Custom user Y axis
26 "⊢----+----+"	
27 "⊢----+----+"	
28 ACA	
29 PRBUF	
30 RTN	
31♦LBL 02	Test for 'plot completed?'
32 RCL 09	
33 RCL 08	
34 X<=Y?	
35 RTN	
36 24.027	Prints sorted plot data
37 PRREGX	
38♦LBL 03	
39 ADV	
40 RDN	
41 STO 09	Resets Xmax to next X
42 SF 07	
43 4	# functions
44 END	

DIAGNOSTIC TESTS
OF AP1, 2, 3, 4
WITH AXD
FOR SELECTED X VALUES

INPUT:

R00=	-36.857	Ymin
R01=	0.857	Ymax
R02=	133.000	Plot field width
R03=	3.500	Desired scale factor
R04=	"AXD"	Custom Y-axis routine
R08=	0.000	Xmin
R09=	0.000	Xmax
R10=	1.000	Delta X
R12=	0.3645	Symbol order
R15=	"AP1"	1st function
R16=	"AP2"	2nd function
R17=	"AP3"	3rd function
R18=	"AP4"	4th function

```
Selects dB pl. CF 00
Symbol order   SF 04
Custom Y axis  SF 09
# func.   4.000 ENTER↑


            XROM "MP"
' AP1
: AP2
: AP3
' AP4
Y: -36.857 TO 0.857
X: 0.000 TO 0.000
ΔX=1.000

   ANT PAT/UNIF ILLUM
          DB
 -40  -30  -20  -10   0
   +----+----+----+----+
  0.0                 '
   +----+----+----+----+

R24=  130.020
R25=  130.028
R26=  130.042
R27=  130.062
```

```
                XROM "MP"
    +----+----+----+----+
 1.0                  :0
    +----+----+----+----+

R24=  118.020
R25=  125.062
R26=  128.042
R27=  129.028

             3.000 STO 08
                   STO 09
                   STO 10
                      RDN
                XROM "MP"
    +----+----+----+----+
 3.0      '      :  : '
    +----+----+----+----+

R24=  37.062
R25=  84.020
R26=  106.042
R27=  118.028

                XROM "MP"
    +----+----+----+----+
 6.0      '      :
    +----+----+----+----+

R24=  36.020
R25=  37.028
R26=  37.062
R27=  79.042
```

Figure 29. MP diagnostic plots of the 4 functions of Example 24 made with the MP diagnostic routine of Example 26. The values are plotted for the selected values of X and the sorted plot data are then printed. The first plot has a custom user header, scale units, linear Y-axis scale, and custom linear Y axes. The others have only the custom linear Y axes. Execution times: 45 sec for X=0; 37 sec each for others.

<u>Example 27.</u> Plot the set of 4 functions used in Example 24 in the same manner as was done in Example 26, but clear flag F21 after the initial custom Y axis. Reset F21 to print the plot data in the MP print buffer registers after each line. Print the value of X for each set of plot data so as to identify it. Use the same values of X as were selected for Example 26.

The 'AXC' routine shown below is the same as 'AXD' except for the changes made in LBL 02 to accomplish the operations specified above. The X-axis labels used with 'AXC' are produced by the 'XL1' routine as in Example 26.

The results for Example 27 are shown in Figure 30 for the 4 selected values of X. It is seen that even though no plot symbols were printed, the sorted plot data are still available.

APPLICATION PROGRAM FOR: MP

Program	Comments
01♦LBL "AXC" 02 RCL 08 03 X=0? 04 XEQ 00 05 XEQ 01 06 XEQ 02 07 RTN	Custom user Y axis sorted plot data routine for dB plots Alternative form to GTO's of 'XL1'
08♦LBL 00 09 ADV 10 " ANT PAT/" 11 "⊢UNIF ILLUM" 12 ACA 13 PRBUF	Custom header
14 " " 15 "⊢ DB" 16 ACA 17 PRBUF	Scale units
18 " -40" 19 "⊢ -30 -20" 20 "⊢ -10 0" 21 ACA 22 PRBUF 23 RTN	Custom user Y-axis scale
24♦LBL 01 25 " +" 26 "⊢----+----+" 27 "⊢----+----+" 28 ACA 29 PRBUF 30 RTN	Custom user Y-axis
31♦LBL 02 32 RCL 09 33 RCL 08 34 " X = " 35 ACA 36 ACX 37 PRBUF 38 CF 21 39 X<=Y? 40 RTN 41 SF 21 42 24.027 43 PRREGX	 Prints value of X Clears printer enable flag Test for "done?" Resets printer flag Prints sorted plot data
44♦LBL 03 45 ADV 46 RDN 47 STO 09 48 SF 07 49 4 50 END	 Resets Xmax to next X # functions

```
SORTED PLOT DATA
FOR AP1, 2, 3, 4
WITH AXC
FOR SELECTED X VALUES

INPUT:

R00= -36.857     Ymin
R01=  0.857      Ymax
R02=  133.000    Plot field width
R03=  3.500      Desired scale factor
R04= "AXC"       Custom Y-axis routine

R08=  0.000      Xmin
R09=  0.000      Xmax
R10=  1.000      Delta X

R12=  0.3645     Symbol order
```

```
R15= "AP1"   1st function
R16= "AP2"   2nd function
R17= "AP3"   3rd function
R18= "AP4"   4th function

Selects dB plot  CF 00
Symbol order     SF 04
Custom Y axis    SF 09
# functions 4.000 ENTER↑

            XROM "MP"
 : AP1
 : AP2
 : AP3
 : AP4
Y: -36.857 TO 0.857
X: 0.000 TO 0.000
ΔX=1.000

  ANT PAT/UNIF ILLUM
        DB
 -40  -30  -20  -10   0
  +----+----+----+----+
 X =  0.000

R24=  130.020
R25=  130.028
R26=  130.042
R27=  130.062

            XROM "MP"
  +----+----+----+----+
 X =  1.000

R24=  118.020
R25=  125.062
R26=  128.042
R27=  129.028

          3.000 STO 08
                STO 09
                STO 10
                   RDN
             XROM "MP"
  +----+----+----+----+
 X =  3.000

R24=  37.062
R25=  84.020
R26=  106.042
R27=  118.028

             XROM "MP"
  +----+----+----+----+
 X =  6.000

R24=  36.020
R25=  37.028
R26=  37.062
R27=  79.042
```

Figure 30. MP sorted plot data for the 4 functions of Example 24 obtained with the MP diagnostic routine of Example 27, which clears F21 to suppress printing of the plot symbols. The sorted plot data are the same as those obtained in Example 26. Execution times: 45 sec for X=0; 37 sec each for others.

J.6.3. Custom User Plot Routine.

MP can be used to produce custom plots with symbols other than the standard single-column symbols normally used. To do this, the MP control program 'MPC' is used to execute MP on a line by line basis to generate and sort the function values to be plotted without printing the standard output, as was illustrated in Example 27 of Section J.6.2 for a single line.

After MP is exited with the sorted plot data for each line, a custom user plot routine is used to produce whatever special plot symbols the user may desire and to skip the proper number of columns between symbols. Tests must be included for symbol overlap, which can either control the plotting of partial symbols or suppress their plotting according to an established symbol precedence. 'PL1' is a typical custom user plot routine that includes the printing of partially overlapped symbols.

After each line is plotted by the custom user plot routine, control is returned to 'MPC' which then proceeds in the same manner for the next value of X until the symbols for Xmax have been plotted. The custom user Y axis is then plotted and execution stops. The full exposition of custom user plot routines can not be presented here because of space limitations, but the interested user can explore the procedure outlined above to produce a wide variety of custom plots. Special symbols can be selected appropriate to the application and may range in size from several different 3x3 symbols to an even greater choice of 5x5 and 7x7 symbols.

LINE BY LINE ANALYSIS OF MP

See the HP writeup for the complete routine listing and line by line analysis, since MP and HP share the same program lines.

REFERENCES FOR MP

See PPC Calculator Journal, V7N1P29c, V7N9P17b and V7N10P11a.

CONTRIBUTORS HISTORY FOR MP

The origin of multifunction plotting dates back to January 1980, when the first 3-function plotting routines appeared in the PPC Calculator Journal by Jake Schwartz (1820) and Jim DeArras (4706). A more streamlined version to handle up to 4 single-column plot symbols simultaneously was later written by Phil Fraundorf (1025) and this was expanded to handle up to 7 functions. At the same time the high resolution plotting routine was also being developed for the ROM. They both performed many of the same type of calculations, and it was Tim Fischer (5793) who first suggested merging the two routines into one. The version in the ROM is primarily Tim's creation from the ground up. It was easier to redesign the two functions together from scratch than to attempt to simply merge them as they existed at that time. Additional features were added by Tim as suggestions arrived in the mail and by phone, until the final version was achieved, around May 1,1981.

The following people had some part in the development, testing and/or documentation of the MP routine: Charles Allen (4691), John Burkhart (4382), Cliff Carrie (834), John Dearing (2791), Tim Fisher (5793), Phil Fraundorf (1025), Bill Hermanson (4115), Roger Hill (4940), Frits Kuyt (236), Clinton Lew (5573), Jake Schwartz (1820), Jack Sutton (5622), Hans-Gunter Lutke Uphues (5286), Steve Wandzura (4635), Larry Weisenberger (1793) and William Wimsatt (5807).

FINAL REMARKS FOR MP

This routine is extremely powerful because of the dozens of options open to the user. As a result, we have only scratched the surface in this fairly lengthy discussion of MP. Many more uses for this routine will undoubtedly be discovered in the future by the thousands of ROM users, and hopefully they will be willing to share their applications with the remainder of the members through articles for PPC Calculator Journal.

FURTHER ASSISTANCE ON MP

Contact Jake Schwartz (1820) at 7700 Fairfield Street, Philadelphia, Penna. 19152 (home phone 215-331-5324); or Tim Fischer (5793) at 7475 Morgan Rd., Bldg 11 Apt 13, Liverpool,N.Y. 13088 (home phone 315-457-6079).

TECHNICAL DETAILS

XROM: 20,28	MP	SIZE: 035

Stack Usage:
- 0 T:
- 1 Z: ALL USED
- 2 Y:
- 3 X:
- 4 L:

Alpha Register Usage:
- 5 M: USED
- 6 N: USED
- 7 O: USED
- 8 P: NOT USED

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED
- 15 e: NOT USED

ΣREG: NOT USED

Data Registers:
- R00: USED
- R01 to R05: USED
- R06: USED
- R07: USED
- R08: USED
- R09: USED
- R10: USED
- R11: USED
- R12: USED
- R13 to R34: USED

Flag Usage:
- 04: Change symbol usage
- 05: Plot overflow mode
- 06: Plot overflow mode
- 07: Skip standard header
- 08: Used internally
- 09: Print custom Y axis
- 10: Set for value plotting; clear for function plotting
- 25: NOT USED

Display Mode: ANY

Angular Mode: NOT USED

Unused Subroutine Levels: 3

Global Labels Called:

Direct	Secondary
IF	User Y-axis routine, User functions in RAM

Local Labels In This Routine:

00 (10 times), 01-12, 13 and 14 (twice each), 15-19, 24, 25

Execution Time:
See section A.3 Execution Times for MP .

Peripherals Required: 82143A Printer

Interruptible? YES, but slows it down slightly
Execute Anytime? NO

Program File: MP

Bytes In RAM: 596

Registers To Copy: 86

Other Comments:
Routine clears flag 55 by executing IF in order to speed up execution. Flag register d, with F55 set, is stored in 0 and restored when printing is done.

NOTES

APPENDIX I - ILLUSTRATIONS & FIGURES

END

MS - MEMORY TO STACK

This routine will store (copy) five registers (from memory) into the "stack" registers X thru T and L. The use of the term "stack", as used here, includes LAST X - the L register. HP does not consider L to be part of the stack when CLST or PRSTK is executed. MS, and its inverse routine, SM will store the "stack" contents in a contiguous block of registers, n to n + 4, starting with register n, which is stored in R06. (n is positive integer)

Example 1: Registers 1 thru 5 are stored into the "stack" by the following.

```
1  STO 06
XEQ MS
```

Example 2: The five registers starting with register 7 are to be recalled to the "stack". This is done by:

```
7  STO 06
XEQ MS
```

COMPLETE INSTRUCTIONS FOR MS

MS uses register 06 to provide the lowest register of a contiguous block of five registers. Any number, n, may be used (stored in R06) provided (n ≤ SIZE)-5 and SIZE ≥ 7. The control number in R06 is maintained (unchanged after routine is finished) for the convenience of repeated use of MS and SM without the need to store the control number in register 06 each time. The R06 control number (which may also be thought of as a pointer) may be zero provided that the XROM MS call is followed by RCL IND 06. This extra two byte instruction is required to correct an error in loading the routine in the ROM. Line 166 (the last DSE 06 instruction) should have had a NOP following it.

WARNING: If Register 06 is zero follow the XROM MS call with RCL IND 06.

In normal use R06 should (must) contain the positive integer 1, or any number from 7 thru 314. Zero is usable provided the WARNING above is followed. The values 2 thru 6, if used would include the control register 06 and this is not normally recommended.

MORE EXAMPLES OF MS

Example 3: Dan, a member of the training department staff, is asked to demonstrate the operation of MS. After a few minutes of thought he decides that the 82143A printer could be used to illustrate the operation of MS (and SM). Dan's program and the output it produced is shown in the box below.

Several useful training examples are included in the program. When Dan shows this DEMO he points out that the two letter ROM labels make nice "DEMO" labels since they fit the seven character limit. Other ROM routines are used to show the condition of registers 0 thru 10 (a convenient minimum byte range used in lines 16, 25, and 38). The "stack" is to be loaded by the student prompted by lines 28 thru 32.

APPLICATION PROGRAM FOR: MS

```
01♦LBL "MS DEMO"
02 ADV
03 ADV
04 SF 12
05 " MS/SM DEMO"
06 PRA
07 CF 12
08 ADV
09 "PRELOAD REGI"
10 "⊢STERS 0-10"
11 PRA
12 "WITH PI.  "
13 "⊢R6 = 1."
14 PRA
15 ADV
16 .01
17 ENTER↑
18 PI
19 ENTER↑
20 0
21 XROM "BI"
22 FIX 2
23 3
24 STO 06
25 .01
26 XROM "BV"
27 ADV
28 "FILL STACK - "
29 "⊢XYZTL  R/S"
30 PRA
31 CLD
32 STOP
33 XEQ 01
34 "NOW DO SM"
35 PRA
36 ADV
37 XROM "SM"
38 .01
39 XROM "BV"
40 ADV
41 "CLEAR STACK"
42 PRA
43 CLST
44 STO L
45 XEQ 01
46 "NOW DO MS"
47 PRA
48 XROM "MS"
49♦LBL 01
50 PRSTK
51 "L=  "
52 ARCL L
53 PRA
54 CLD
55 ADV
56 END
```

BI and BV are used as described in their respective sections. The "stack" is listed using LBL 01. (line 33) LBL 01 is needed for this purpose, because the PRSTK at line 50 doesn't include L. Lines 51 thru 53 compensate for this. A similar situation occurs for the CLST at line 43. Line 44 clears L.

Example 4: PPC Member, Henry, doesn't like the recommendation that 2 thru 6 not be used with MS and SM. After all, why should there be limits? After studying the routine listing, Henry notices that R06 continuously decrements from N + 4 to N +3, n +2, n + 1, and finely back to n upon completion of the routine. What should be in the stack so that R06 is always correct? Writing down a stack analysis seemed like too much work, so Henry pressed a few buttons and noticed the following. MS and SM work fine if R06 contains 2 thru 6 provided that the number 6 was in the correct position of the stack. Henry made a table for future reference.

TABLE MS-1

IF R06 Contains:	"6" Must Be in:
2	L
3	T
4	Z
5	Y
6	X

The overlapping of data storage with the pointer register saves one register (7 bytes), but the cost of placing the digit 6 in the correct stack register may require 1 to 3 bytes for a net gain of 6 to 4 bytes.

LINE BY LINE ANALYSIS OF MS

MS is the inverse of SM. In SM the L register is stored last. In MS L is recalled first, followed

T E C H N I C A L   D E T A I L S

XROM: 10,48 | **MS** | SIZE: ≥ 005*

Stack Usage:
- 0 T: USED
- 1 Z: USED
- 2 Y: USED
- 3 X: USED
- 4 L: USED

Alpha Register Usage:
- 5 M: NOT USED
- 6 N: NOT USED
- 7 O: NOT USED
- 8 P: NOT USED

Other Status Registers:
- 9 Q:
- 10 ⊦:
- 11 a: NONE USED BY
- 12 b: ROUTINE
- 13 c:
- 14 d:
- 15 e:

ΣREG: NOT USED

Data Registers:
- R00:
- R06: Lowest Register of
- R07: block.**
- R08:
- R09:
- R10:
- R11:
- R12:

Flag Usage: NONE
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10:
- 25:

Display Mode: N/A

Angular Mode: N/A

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: 1.3 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	BL
Bytes In RAM:	31
Registers To Copy:	46

Other Comments:

*Minimum size is value in R06 + 5.

**If R06 = 0 see text for "BUG" information.

by T, Z, Y, and X. Lines 156 and 157 increase the pointer value in R06 by 4. "L" is recalled and stored by lines 158 and 159. R06 is decremented and used for another indirect recall by lines 160 and 161. This operation is repeated three more times by lines 161 thru 167. The DSE 06 at line 166 will skip the last RCL if R06 initially contained zero. The correct function could be performed if a NOP followed line 166. This was accidentally omitted when the ROM was loaded. See warning above. MS ends at line 168.

Routine Listing For: MS	
155◆LBL "MS"	162 DSE 06
156 4	163 RCL IND 06
157 ST+ 06	164 DSE 06
158 RCL IND 06	165 RCL IND 06
159 SIGN	166 DSE 06
160 DSE 06	167 RCL IND 06
161 RCL IND 06	168 END

CONTRIBUTORS HISTORY FOR MS

See SM.

FINAL REMARKS FOR MS

See SM.

FURTHER ASSISTANCE ON MS

Richard Nelson (1) (714) 754-6226 P.M.
Richard Schwartz (2289) (213) 447-6574.

NOTES

MT

MT - MANTISSA OF X

MT is a mantissa function that operates much like a built-in function (such as SIGN).

Example 1: Find the mantissa of 1/π. Key π, 1/x, XEQ MT. The result is 3.183098861.

COMPLETE INSTRUCTIONS FOR MT

MT is used like any other single-argument mathematical function available in the HP-41. With an argument in the X register, XEQ MT will replace that argument with its mantissa. The argument will be placed in LSTX and the rest of the stack will be left unchanged.

There are times when one wishes to extract, for observation or manipulation, the mantissa of a number stored in the X register of the HP-41. For example, one application might be to display the full ten significant figures of a number which is so big or small that the HP-41 shifts automatically into scientific notation where only eight of the significant digits can be observed. Notice that when in FIX9, SCI9, or ENG9 mode the eight displayed digits of numbers with exponents larger than 9 or smaller than -9 are not rounded. In other modes, however, the eighth digit is rounded. Unless one knows the mode that the calculator is in, it is not obvious whether the observed number is rounded or not.

One may attempt to construct an MT function by dividing the argument by (10 raised to the INT(LOG (ABS(of the argument)))). However, rounding errors can produce answers which are in error by a factor of 10 (try this on 9.9999999 15, for example.) Further, a routine which uses LOG and 10↑X will be slow since these alone have a combined execution time of about 350ms.

The PPC ROM routine MT uses byte manipulation in the alpha register to extract the mantissa of a number initially in the X register. This technique avoids rounding errors as well as time consuming math functions. Recalling that numbers are stored in the format Sn.nnnnnnnnnSee (see *PPC CALCULATOR JOURNAL*, V6N6P19d) one observes that trimming two bytes (four digits) from the right end trims one too many digits and the least significant digit of the mantissa is lost. Trimming only one byte from the right allows us to set the exponent to zero while keeping the full mantissa. This is acceptable if the exponent of the argument was initially zero (positive). However, if the exponent was initially negative, then the sign digit of the exponent is 9, not 0, and replacing the number part of the exponent with zeros produces a number with the correct mantissa, but with an exponent of -100! This difficulty may be overcome by noting when the exponent of the argument is negative and replacing the argument's exponent with a hexadecimal A0 instead of placing a zero in the most significant digit of the exponent and carrying the 1 over into the sign digit which also becomes a 0 (9+1=0, carry a 1). The additional carry is discarded.

LINE BY LINE ANALYSIS OF MT

MT begins by clearing the ALPHA register, thus setting registers M, N, O and P to zero. The argument, initially in X, is stored in M. ASTO M shifts the argument one place to the right in the M register, clipping off the two digits of the exponent. Performing an INT on the argument, which is also still in the X register, performs two functions. First, the

TECHNICAL DETAILS

XROM: 10,28 | **MT** | SIZE: 000

Stack Usage:
- 0 T: UNCHANGED
- 1 Z: UNCHANGED
- 2 Y: UNCHANGED
- 3 X: Mantissa (x)
- 4 L: X

Alpha Register Usage:
- 5 M:
- 6 N: hex 10 and 7
- 7 O: nulls
- 8 P:

Other Status Registers: NONE USED
- 9 Q:
- 10 ⊢:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: .5 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	VM
Bytes In RAM:	26
Registers To Copy:	60

Other Comments:

argument is stored into LSTX. Second, if the exponent of the argument is negative the result of an INT will be zero. If the exponent is positive or zero, a 00 (hex) is appended to the data in the M register. If the exponent is negative, an A0 (hex) is appended to the M register instead. Adding zero to the M register, which now contains the answer, will do nothing if the exponent was initially non-negative. However, it will cause the A in the exponent to normalize the 9 in the exponent's sign digit if the exponent was initially negative. The routine ends by swapping the zero in X for the answer in M.

CONTRIBUTORS HISTORY FOR MT

Mantissa routines have generally used the LOG function. Roger Hill (4940) and Eric Barsalou (4304) used synthetic programming in *PPC CALCULATOR JOURNAL*, V7N7P18a and V7N8P2d, respectively, to write faster routines. The ROM version of MT was written by Dave Kaplan (3678). It is one of the fastest ROM routines, and represents a vast improvement over earlier versions.

FURTHER ASSISTANCE ON MT

Call David R. Kaplan (3678) at (703) 250-6621.
Call Keith Kendall (5425) at (801) 967-8080.

Routine Listing For: **MT**

```
29♦LBL "MT"          36 X=0?
30 CLA                37 "├"
31 STO [             38 CLX
32 ASTO [            39 ST+ [
33 INT               40 X<> [
34 X≠0?              41 RTN
35 "├♦"
```

NOTES

NC - N TH CHARACTER

NC is an ALPHA register editing subroutine. NC replaces a string of up to 24 bytes with one of the ten rightmost bytes. The extracted byte is stored in the X register as a single alpha character. The integer part of the one through ten number in the X register designates the position number of the byte to be extracted. Bytes are counted from right to left and may be of any type, such as a number or an alpha character. NC preserves the integer part of X in LASTX, and the user data in Y and Z are preserved. NC makes no subroutine calls, uses no data registers and executes in 1.5 seconds. Flag 25 is cleared but all other states are preserved.

Example 1: Byte Extraction

Extract the leftmost byte of a six byte alpha string.

DO:	SEE:	RESULTS:
AON, ABCDEF	ABCDEF_	alpha string
AOFF, 6	6_	position # of A
XEQ NC	A	alpha character
AON	A	byte extracted

Example 2: Digit Decoding

Determine the number of displayed digits.

DO:	SEE:	RESULTS:
CLA		clear ALPHA
RCL d	NNN	flag register
STO M	NNN	stored in ALPHA
RDN, 3	3_	position number
XEQ NC	one byte	byte extracted
RDN, XEQ CD	integer	decimal of byte
16, MOD	0 thru 15	number of digits

COMPLETE INSTRUCTIONS FOR NC

NC extracts one of the ten rightmost bytes of up to 24 bytes in ALPHA. NC produces two outputs. First, a non-normalized number appears as a single alpha character in ALPHA. Second, the extracted byte is stored as alpha data in the X register.

To use NC, place the data containing the byte to be extracted into the alpha register. Numbers and non-normalized data are stored and recalled from ALPHA using such instructions as STO M, RCL M and X<>M. Then, place a number (one through ten) in X. The integer part of the number represents the position number of the byte to be extracted. XEQ NC to extract the byte.

NC preserves the integer part of X in LASTX, and the user data in Y and Z are preserved. NC leaves a copy of the current contents of d in T. NC makes no subroutine calls, uses no data registers and executes in 1.5 seconds. NC may be SST'ed, however, do not use the printer instruction PRSTK when SST'ing due to normalization of the stack.

APPLICATION PROGRAM 1 FOR NC

This brief creation of William Cheeseman (4381) is an alternate version of NC that is restricted to 6 characters or less in alpha and 1 ≤ x ≥ 6.

```
LBL "NC6"
LBL 00
ASTO M
DSE X
GTO 00
ASHF
ASTO X
RTN
```

APPLICATION PROGRAM 2 FOR NC

This alternate version of NC was written by Richard Chandler (6152). It arrived during the ROM loading. It selects the xth character (1 ≤ x ≤ 12) from the left of a string of up to 24 characters. The counting of characters from the left may be more useful for some applications. The former Y and Z end up in X and Y, but the display format is changed (usually). Rather astoundingly, this program contains no synthetic instructions whatsoever.

APPLICATION PROGRAM FOR: NC

```
01♦LBL "NC"
02 6
03 X<Y?
04 GTO 10
05 ASTO L
06 CLA
07 ARCL L
08 RDN
09 GTO 11

10♦LBL 10
11 ASHF
12 -

13♦LBL 11
14 9
15 -
16 XEQ 14
17 ASHF
18 ASTO L
19 CLA
20 "*"
21 ARCL L
22 ASHF
23 RDN
24 RTN

25♦LBL "SU"
26 6
27 X<Y?
28 GTO 12
29 RDN
30 ASTO T
31 ASHF
32 ASTO Z
33 CLA
34 ARCL T
35 XEQ 13
36 ARCL Y
37 RTN

38♦LBL 12
39 -
40 ASTO Z
41 ASHF
42 XEQ 13
43 ASTO L
44 CLA
45 ARCL Y
46 ARCL L
47 RTN

48♦LBL 13
49 9
50 -
51 XEQ 14
52 12
53 +
54 ASHF
55 ASTO T
56 ASHF
57 ASTO L
58 CLA
59 "*"
60 ARCL T
61 ASTO T
62 CLA
63 ARCL T
64 XEQ 14
65 ASHF
66 ASHF
67 ARCL Y
68 ARCL L
69 RDN
70 RTN

71♦LBL 14
72 FIX IND X
73 ASTO T
74 CLA
75 ARCL X
76 ARCL T
77 .END.
```

LINE BY LINE ANALYSIS OF NC

NC and SU use common program lines and use Flag 25 to skip lines and branch as required. NC initially clears Flag 25 at line 173. Lines 174 and 177 bypass SU entry. Line 178 takes the integer part of X for the position number, which is simultaneously subtracted from ten and stored in LASTX at line 181. The difference is used to indirectly set the number of display digits for the ARCL sequence of lines 182 through 186. Line 184 appends a variable number of bytes, from 4 through 13, to the original string in ALPHA. A position number of one will append 13 bytes and a ten

Routine Listing For: **NC**

```
172♦LBL "NC"          202 DSE L
173 CF 25             203 CLX
174 GTO 14            204 X<> L
177♦LBL 14            205 10↑X
178 INT               206 RCL d
179 E1                207 FIX 0
180 X<>Y              208 CF 29
181 -                 209 ARCL Y
182 RCL d             210 STO d
183 SCI IND Y         211 RDN
184 ARCL Y            212 CLX
185 STO d             213 ISG L
186 RDN               214 CLX
187 X<> ]             215 X<> ↑
188 FS? 25            216 STO \
189 RCL ↑             217 CLX
190 "⊢*"              218 X<> ]
191 FC?C 25           219 STO [
192 GTO 14            220 RDN
193 X<> Z             221 RTN
194 STO ]
195 "⊢******"         222♦LBL 14
196 X<> Z             223 X<> ]
197 STO ↑             224 CLA
198 RDN               225 STO [
199 X<> ]             226 ASTO X
200 X<> \             227 END
201 STO [
```

will append 4 bytes. The result is that all the bytes to the left of the byte to be extracted are now in P and O, and the byte to be extracted is in the left-most position in N. The RCL d and STO d instructions at lines 182 and 185, respectively, preserve the original display status. Line 187 places six NULLs immediately to the left of the byte to be extracted. Line 189 is skipped by **NC** and executed by **SU** for valid inputs in X. Line 190 shifts the byte to be extracted into the rightmost position in O; O now contains only the byte to be extracted. Lines 191 and 192 advance the program pointer to line 222. Line 223 stores the contents of O, containing the byte to be extracted, in X. ALPHA is cleared at line 224, and the extracted byte is stored in ALPHA at line 225. The non-normalized number in ALPHA is stored as alpha data into X at line 226.

REFERENCES FOR NC

Wickes, William C. "Synthetic Function Routines." *PPC CALCULATOR JOURNAL*, V7N3P7 (April 1980).

Wickes, William C. *SYNTHETIC PROGRAMMING ON THE HP-41C.* (Larken Publications, 1980), page 64.

CONTRIBUTORS HISTORY FOR NC

William C. Wickes (3735) presented one of the first practical applications of synthetic programming with his ISO. HM, his program version of the familiar word-guessing Hangman game, effectively demonstrates a use of ISO.

Carter P. Buck (4783) wrote the **NC** / **SU** integrated subroutines of the PPC Custom ROM. These are improved versions of Wickes' ISO and SUB.

FURTHER ASSISTANCE ON NC

Call Carter Buck (4783) at (415) 653-6901
Call Richard Chandler (6152) at (919) 851-2153.

TECHNICAL DETAILS

XROM: 10,38 | **NC** | SIZE: 000

Stack Usage:
- 0 T: d (flags)
- 1 Z: Z
- 2 Y: Y
- 3 X: character
- 4 L: X

Alpha Register Usage:
- 5 M:
- 6 N: nth character
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: ONLY FLAG 25 USED
04:
05:
06:
07:
08:
09:
10:
25: CLEARED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
PART OF **SU**	NONE

Local Labels In This Routine:
NONE

Execution Time: 1.3 seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: **VK**
Bytes In RAM: 112 INCLUDING **SU**
Registers To Copy: 63

Other Comments:

NH - NNN TO HEX

This routine is the ROM version of "DECODE". It translates a non-normalized number (NNN) in the X register into an equivalent 14-byte hexadecimal expression in the Alpha register, leaving the 41C in ALPHA mode so that the translated NNN is displayed. The original NNN is preserved in X and the contents of Y and Z are also saved. Two options are available: If flag 10 is clear, the result is displayed in standard hex notation; if flag 10 is set, execution is much faster, and the result is desplayed in "natural (or machine) language." The values 0 through 9 are the same in either case; the others are:

Hex Value	Standard Notation	"Natural Language"
10	A	:
11	B	;
12	C	<
13	D	=
14	E	>
15	F	?

Example 1: Ascertain the current contents of register c.

Do	See
RCL c	?.⊠001 >>
XEQ NH	0FA001690EF0EE

This shows the configuration of the 41C in default status. The "0FA" (decimal 250) is the absolute address of "ΣREG", the beginning of the statistics block. The "00" are 2 nybbles used for printer scratch, empty at the moment. The "169" is the "cold" start constant". The 0EF (decimal 239) is the "curtain" or absolute address of register 00. Note that "ΣREG" is 11 registers greater because register 11 begins the statistics block in default status. The "0EE" (decimal 238) is the location of .END., the permanent end mark. Because there are no RAM programs in memory, the .END. lies in the register immediately below register 00.

Do	See
ALPHA (Off)	?.⊠001 (the original NNN
SF 10	?.⊠001 is still in X)
XEQ NH	0?:001690>?0>>

This is the same thing in "natural language".

COMPLETE INSTRUCTIONS FOR NH

While NH preserves the original NNN in X and saves the contents of Y and Z, it destroys T. Although it uses register d, it restores the original flag settings. If flag 10 is set to produce the faster "natural language," it will not be cleared by NH itself. No other flags are used. NH leaves the 41C exactly as its inverse HN expects to find it, regardless of whether flag 10 is set or clear.

Although designed primarily to decode NNN's, particularly those recalled from the status registers or from program steps in RAM or ROM, it should be emphasized that NH will faithfully decode anything which it finds in the X register, be it a number, an alpha expression or any other possible combination of 56 bits. Thus, it is of such general application that further examples are almost superfluous; those following are merely illustrative.

MORE EXAMPLES OF NH

Example 2: Quickly check the flag status without a printer.

We wish to run a program where the setting of flags 05-20, which are not enunciated, is critical.

Do	See
ALPHA (Off)	?.⊠001 >>
RCL d	0.2000
XEQ NH	0020002<048000

Each of the 14 bytes displayed represents 4 flags so we see immediately that flags 00 through 07 are clear, but that flag 10 is still set from the previous example.

ALPHA (Off)	0.2000
CF 10	0.2000
RCL d	0.0000
XEQ NH	0000002C048000

The 6 initial 0's tell us that the first 24 flags (00-23) are now clear and we may proceed to run our program. This flag setting, incidently, is that of 41C default status: audio enabled, decimal radix, digit grouping and FIX 4.

Example 3: Trace the 41C's internal treatment of numbers.

Do	See
ALPHA (Off)	0.0000
PI	3.1416
SF 10	3.1416

(Since we are dealing here with a number, the faster flag 10 option is indicated.)

XEQ NH	03141592654000

We see the internal maintenance of Pi: 10 full digits of precision; no rounding; no decimal point; exponential notation.

ALPHA (Off)	3.1416
EEX 4	1 4
*	31,415.9265
XEQ NH	03141592654004

Although the number and its display have changed significantly, internally the 41C--with great economy of effort--has changed only one digit; as a matter of fact, it has changed just one of the 56 bits in register X.

ALPHA (Off)	3,415.9265
PI	3.1416
RND	3.1416
XEQ NH	03141600000000

Although the display was not changed at all by RND, internally the 41C has altered the number significantly, rounding one, and dropping five, digits of precision.

Example 4: Decipher an END statement.

This requires that we first go to a RAM program and, immediately before the END, insert X≠0?, an instruction which provides the correct byte jumper controller for a 3-byte jump, but which will not affect the RAM program's execution, because END statements are not skipped by immediately preceding conditionals under any circumstances. Then SST to the END, PACK and

Do	See	
PRGM (Off)	3.1416	
BYTE JUMP (See	3.1416	(The END is in M)
RCL M Appendix G)	0.0000 -91	(The END is in X)
XEQ NH	00000000CA0409	

The "C" indicates a global instruction from row 12 of the hex table which may be either a global LBL or an END statement, the "A04" is binary 1010 000 0100. Regrouping this as 101 000000100 gives us, in binary, the distance back to the next preceding global LBL or END in RAM memory. The group of 9 bits shows the number of whole registers (in this case 4) and the group of 3 bits shows the number of additional bytes (in this case 5). Therefore, the next preceding global instruction is 4 registers and 5 bytes (or a total of 33 bytes) back in the global address chain. The last two digits would be "Fn" if this were a global LBL; the fact that they are not shows this to be an END. The 0 indicates a non-permanent END and would be a 2 if this were the permanent .END. The "9" shows that the file is PACKed and would be a "D" if it were not PACKed.

Example 5: The NH routine only uses one stack register. This feature may be used to "decode" text lines, such as those found in the LG routine. SST lines 02 & 03 of LG to produce a 21 character alpha string. Run the following program to "decode" the alpha register.

```
01   LBL "A-HEX"
02   RCL M       Lines 2,3, &4 fill stack with
03   RCL N       three seven character alpha
04   RCL O       text strings.
05   XROM 'NH    Produces 11C2E47C3C7AF1.
06   XROM 'VA    Prints line if printer plugged on.
07   RDN         Position next seven characters.
08   XROM 'NH    Produces 11663E1E3D78F9.
09   XROM 'VA'   Prints line if printer plugged in/on.
10   RDN         Position last seven characters
11   XROM 'NH    Produces 119E1D9BBF4E87.
12   XROM 'VA    Prints line if printer plugged in/on.
13   AOFF
14   RTN
```

See similar example for the CD routine. STOP may be placed following lines 12, 09, and 06 if a printer is not used. Don't forget the addition of TEXT 14, TEXT 8, and APPEND instructions if these text lines are going to be placed into program memory using LB.

APPLICATION PROGRAM 1 FOR NH

Because NH produces 14 bytes of information in the alpha register the first 2 bytes rapidly scroll out of view, sometimes making it necessary to press ALPHA, ALPHA to check them. This minor difficulty can be overcome with a RAM access routine which positions leading spaces, but this in turn makes the process annoyingly slow when only the program pointer address from register b is needed. The following RAM access program is a suggested way of handling both situations:

```
01  LBL "DECODEb"   07  RCL M                 13  X<>L
02  SF 09           08  "  "   (2 spaces)     14  FS? 09
03  LBL "DECODE"    09  X<>M                  15  ASHF
04  XROM  NH        10  STO O                 16  FS?C 09
05  SIGN            11  RDN                   17  ASHF
06  X<>N            12  STO N                 18  TONE 9
                                              19  END
```

For a full 14-byte display, XEQ "DECODE" and the scrolling results will be signalled by a tone and then preceded by 2 spaces, making the viewing easier. For a display of just the rightmost 4 bytes following RCL b, XEQ "DECODEb", and only the program pointer address will be displayed. Except for the possible spaces in register O, this routine leaves the 41C in the same configuration as NH itself, or exactly as the inverse HN (which ignores these spaces) expects to find it.

Routine Listing For: NH

```
01♦LBL "NH"        32 FC? 04
02 CLA             33 GTO 14
03 STO [
04 SIGN            34♦LBL 13
05 X<> d           35 SF 01
06 "├♦♦*"          36 CF 02
07 .               37 CF 03
08 X<> [           38 CF 04
09 "├♦♦"           39 FS?C 07
10 X<> [           40 GTO 14
11 FIX 9           41 SF 07
12 ARCL \          42 FS?C 06
13 "├**"           43 GTO 14
14 ARCL ]          44 SF 06
15 X<> ]           45 CF 05
16 FIX 3
17 ARCL ]          46♦LBL 14
18 STO ]           47 X<> d
19 "├*"            48 STO [
20 RDN             49 "├*"
21 STO d           50 RDN
22 CLX             51 DSE X
23 FS? 10          52 GTO 01
24 GTO 12
25 RDN             53♦LBL 12
26 14              54 STO ↑
                   55 X<> ]
27♦LBL 01          56 X<> \
28 RCL ]           57 STO [
29 X<> d           58 X<> L
30 FC? 06          59 AON
31 FS? 05          60 RTN
```

LINE BY LINE ANALYSIS OF NH

Lines 02-22 convert the NNN into "natural language". This is always done first regardless of the setting of flag 10. The SIGN at line 03 is a 1-byte equivalent of STO L. The sequence at lines 06-19 (particularly the curiously "self-manipulating" instructions ARCL N and ARCL O) serves as an excellent example of the power of synthetic instructions in formatting data entirely within the alpha register. Lines 16 and 17 are a 4-byte equivalent of a conventional leftward "pusher" like "├ABCDE" which takes 7 bytes. In synthetic manipulation, where bytes frequently need to be pushed n positions to the left, this variant will save bytes whenever n is greater than two, so long as some register (in this case O) is known to contain data that will behave as intended in the FIXn, ARCL sequence. The otherwise redundant CLX at line 22 is necessary only for the flag 10 option, in which case lines 25-52 are skipped and total execution time is just slightly over one second. If flag 10 is clear, however, the sequence at lines 27-52 is repeated 14 times under the control of the DSE X loop. On each iteration, one hex byte, already in "natural language," is "positioned at" flags 00-07 of Register d. Lines 30-32 test to see if it is a 0-9 numerical byte; if so, the jump at line 33 is taken. If not, it is a byte that needs to be converted from : ; < = > ? to ABCDE or F, respectively, all of which require lines 35-38. The values corresponding to B, D, and F are identified at line 39 and take the jump at line 40. The values corresponding to A and E are identified at line 42 and take the jump at line 43. Lines 44 and 45 are for the "C" value only. [Note that the LBL 13 at line 34 is, alas, completely superfluous; the final edit of NH obviated the need for this label, but in the rush to get the ROM to Corvallis, the label itself was not deleted.] Lines 46-52 format registers O, N, and M for the next iteration. At line 53, the translated NNN, in either standard notation or "natural language," lies in registers O and N. Lines 54-59 shift it to registers N and M, recall the original NNN from L, and display the result with AON, used here instead of AVIEW so that execution is never halted, regardless of the settings of flags 21 and 55.

CONTRIBUTORS HISTORY FOR NH

The original "DECODE" was conceived by William C. Wickes (3735) and appeared in December 1979 in the *PPC CALCULATOR JOURNAL*, V6N8P29. It, and the other Wickes "black box" programs, helped open the door to synthetic programming by permitting the exploration of status and program registers and by demonstrating the power of the synthetic instructions which the programs themselves contained. Bill published a new version of "DECODE" in March 1980 in PPC CJ, V7N2P35 and by the end of the year had developed further successive improvements which were shared with other members. In May 1980, Valentin Albillo (4747) in Madrid, Spain, published in the PPC CJ, V7N4P28 a version of "DECODE" which he called "U" and which translated successive segments of an NNN into octal. Building upon Valentin's work, John McGechie (3324) in Melbourne, Australia, in August 1980 published in the PPC CJ, V7N6P31 a program called "AN" which, in 74 bytes and just a few seconds, decoded an NNN into what John called "natural language" (also known now as "Australian notation").

Evidently at this point several widely-scattered members, working independently, adopted John's work, further refined it, and tried out the idea of first going into "natural language" and then into standard hex notation. As the PPC ROM neared realization, Synthetic Coordinator, Keith Jarett, was confronted with what he called the "CODE/DECODE Competition." The first "DECODE" routine proposed for the ROM (see PPC CJ, V7N7P19) was called "N↑H" and was essentially John McGechie's "AN". By October 1980, (see PPC CJ, V7N8 P10) this had been superceded by "DCD", a submission by Gerard Westen (4780) which produced standard notation, but was only 113 bytes long. In December 1980, (see PPC CJ, V7N10P16c) Keith reported receiving two new "DECODE" routines from Phillipe Roussel (4367) and one which combined the best of both approaches from Charles Close (3878).

The version finally selected for the ROM was written by Richard H. Hall (4803) and received a final edit by Roger Hill (4940). This is the fastest "DECODE" routine.

FURTHER ASSISTANCE ON NH

Call Richard H. Hall (4803) at (301) 383-1214.
Call Steven Jacobs (5358) at (801) 484-3672.

TECHNICAL DETAILS

XROM: 10,40 — **NH** — SIZE: 000

Stack Usage:	Flag Usage: ONLY FLAG 10 USED
0 T: USED	04:
1 Z: Z	05:
2 Y: Y	06:
3 X: X	07:
4 L: USED	08:
Alpha Register Usage:	09:
5 M: REPLACED BY HEX DIGITS	10: TESTED ONLY
6 N:	
7 O: CLEARED	
8 P: CLEARED	25:

Other Status Registers:	Display Mode: UNCHANGED
9 Q: NOT USED	
10 ⊢: NOT USED	
11 a: NOT USED	Angular Mode: UNCHANGED
12 b: NOT USED	
13 c: NOT USED	
14 d: USED BUT RESTORED	Unused Subroutine Levels:
15 e: NOT USED	6

ΣREG: UNCHANGED	Global Labels Called:
Data Registers: NONE USED	Direct: NONE — Secondary: NONE
R00:	
R06:	
R07:	
R08:	
R09:	
R10:	
R11:	Local Labels In This Routine:
R12:	1, 12, 13, 14

Execution Time: 5.6-10 seconds; WITH FLAG 10 SET: 1.4 seconds.

Peripherals Required: NONE

		Other Comments:
Interruptible?	YES*	*Interruption may halt execution in PRGM MODE. In this case, switch to RUN MODE before pressing R/S to continue.
Execute Anytime?	NO	
Program File:	NH	
Bytes In RAM:	120	
Registers To Copy:	33	

APPENDIX J – PPC CUSTOM ROM LISTING

```
01♦LBL "MK"
02 CF 07
03 SF 09
04 12
05 XROM "VS"
06 FC?C 25
07 PROMPT

08♦LBL 01
09 XROM "LF"
10 STO 09
11  E
12 +
13 X<>Y
14 STO 10
15 ASTO 11
16 DSE Y
17 GTO 07
18 SF 20
19 FC?C 09
20 GTO 13

21♦LBL 02
22 RCL 09
23 XEQ 11
24 "REG FREE: "
25 RCL d
26 FIX 1
27 ARCL Y
28 STO d
29 XROM "VA"
30 TONE 6
31 PSE

32♦LBL 03
33 "PRE↑POST↑KEY"
34 CLST
35 XROM "VA"
36 TONE 7
37 STOP
38 GTO 14

39♦LBL "1K"
40 CF 20
41 SF 07

42♦LBL "+K"
43♦LBL 14
44 STO 08
45 RDN
46 STO 07
47 RDN
48 STO 06
49 CF 09
50 RCL 10
51 SIGN
52 FS? 20
53 X≠0?
54 GTO 01

55♦LBL 13
56 RCL 08
57 INT
58 X=0?
59 FS? 07
60 FC?C 20
61 GTO 02
62 X<0?
63 SF 09
64 ABS
65 STO Z
66 44
67 -
68 ABS
69 2
70 X<Y?
71 DSE T
72 R↑
73 STO Y
74  E1
75 ST/ Z
76 MOD
77 8
78 *
79 ENTER↑
80 CF 08
81 LASTX
82 FS? 09
83 ST+ Y
84 R↑
85 INT
86 X≠0?
87 X>Y?
88 GTO 08
89 R↑
90 +
91 ST+ Z
92 X<>Y
93 X<=Y?
94 CLX
95 X≠0?
96 SF 08
97 +
98 36
99 -
100 X>0?
101 GTO 08
102 FC? 09
103 RCL ↑
104 FS? 09
105 RCL e
106 FC? 08
107 GTO 14
108 STO [
109 "⊢*"
110 X<> [

111♦LBL 14
112 X<> d
113 FS? IND Y
114 GTO 09
115 SF IND Y
116 X<> d
117 FC? 08
118 GTO 14
119 STO [
120 ARCL 10
121 X<> \

122♦LBL 14
123 FC? 09
124 STO ↑
125 FS?C 09
126 STO e
127 "**"
128 FS? 10
129 ARCL 11
130 X<> Z
131 RCL 07
132 RCL 06
133 XROM "DC"
134 XROM "DC"
135 XROM "DC"
136 FS?C 10
137 GTO 14
138 "⊢♦♦♦"
139 ASTO 11
140 SF 10

141♦LBL 14
142 RCL 09
143 RCL 10
144 X<> c
145 RCL [
146 STO IND Z
147 X<>Y
148 X<> c
149 CLST
150 CLA
151 FC? 10
152 ISG 09
153 SF 20
154 FS? 07
155 RTN
156 FS? 20
157 GTO 03
158 "DONE, NO MORE"
159 SF 09
160 GTO 14

161♦LBL 07
162 "NO ROOM"
163 CF 20
164 CLST

165♦LBL 14
166 FS?C 25
167 RTN
168 XROM "VA"
169 TONE 7
170 TONE 3
171 STOP
172 GTO 01

173♦LBL 08
174 "NO SUCH KEY"
175 GTO 14

176♦LBL 09
177 X<> d
178 "KEY TAKEN"

179♦LBL 14
180 CF 09
181 CF 20
182 FS?C 25
183 RTN
184 XROM "VA"
185 TONE 3
186 PSE
187 "KEYCODE?"
188 CLST
189 RCL 08
190 XROM "VA"
191 TONE 7
192 STOP
193 STO 08
194 GTO 01

195♦LBL "F?"
196 XROM "LF"

197♦LBL 11
198 INT
199 LASTX
200 FRC
201  E3
202 *
203 X<>Y
204 .5
205 FC? 10
206 SIGN
207 -
208 -
209 END

01♦LBL "LF"
02 XROM "E?"
03 17
04 -
05  E3
06 /
07 177
08 +
09 XROM "OM"
10 "*8*****"
11 .
12 ENTER↑
13 DSE T
14 GTO 14

15♦LBL 00
16 X<> IND T
17 X=Y?
18 GTO 14
19 X<> [
20 "⊢8"
21 STO \
22 ARCL X
23 RDN
24 RCL \
25 X<> IND T
26 ISG T
27 GTO 00

28♦LBL 14
29 X<> [
30 ARCL X
31 X<> \
32 SF 10
33 X=Y?
34 DSE T
35 CF 10
36 X<> Z
37 X<> c
38 R↑
39 RTN

40♦LBL "CK"
41 XROM "E?"
42 17
43 -
44  E3
45 /
46 177
47 +
48 DSE X
49 RTN
50 XROM "OM"
51 .
52 STO ↑
53 STO e

54♦LBL 06
55 STO IND Z
56 ISG Z
57 GTO 06
58 X<>Y
59 X<> c
60 R↑
61 RTN

62♦LBL "VA"
63 SF 25
64 PRA
65 SF 25
66 FS?C 21
67 CF 25
68 AVIEW
69 FC?C 25
70 SF 21
71 RTN

72♦LBL "UD"
73 CLA
74 ARCL 00
75 ASTO c
76 ΣREG 01
77 RTN

78♦LBL "PK"
79 XROM "E?"
80 17
81 -
82  E3
83 /
84 175
85 +
86 ENTER↑
87 CF 09
88 CF 10
89  E
90 +
91 XROM "OM"
92 .
93 GTO 03

94♦LBL 07
95 FS?C 09
96 X≠0?
97 FS? 09
98 GTO 03
99 X<> [
100 FS?C 10
101 GTO 08
102 SF 10
103 "**"
104 X<> [
105 X<> \
106 ASTO L
107 GTO 03

108♦LBL 08
109 ARCL L
110 X<> [
111 STO \
112 "⊢***"
113 GTO 02

114♦LBL 01
115 X<> IND T
116 SF 25
117 X≠0?
118 FS?C 25
119 GTO 04
120 "****"
121 ARCL c
122 STO [
123 "⊢****"
124 ASHF
125 X<> \
126 X=0?
127 SF 09
128 CLX
129 X<> \
130 FC? 09
131 STO [
132 X<> [
133 ASHF
134 X<> [
135 FC? 09
136 "⊢"
137 FC? 09
138 X=0?
139 GTO 07
140 RDN
141 RCL [
142 STO \
143 ARCL c

144♦LBL 02
145 X<> \
146 X<> IND Z
147 ISG Z

148♦LBL 03
149 ISG T
150 GTO 01

151♦LBL 04
152 FC? 10
153 GTO 05
154 ARCL L
155 "⊢♦♦♦"
156 X<> [
157 STO IND Z

158♦LBL 05
159 RDN
160 X<> c
161 X<>Y
162 ENTER↑
163 GTO 14

164♦LBL "A?"
165 XROM "LF"

166♦LBL 14
167 CLA
168 INT
169 175
170 -
171 .5
172 FC?C 10
173 SIGN
174 -
175 RTN

176♦LBL "DC"
177 INT
178 256
179 MOD
180 LASTX
181 +
182 OCT
183 X<> d
184 FS?C 11
185 SF 12
186 FS?C 10
187 SF 11
188 FS?C 09
189 SF 10
190 FS? 07
191 SF 09
192 FS? 06
193 SF 08
194 X<> d
195 X<> [
196 RCL \
197 "⊢*"
198 X<> ]
199 X<>Y
200 STO \
201 X<> ↑
202 "⊢*"
203 STO ↑
204 RDN
205 X<> ]
206 X<> \
207 STO [
208 RDN
209 END

01♦LBL "ML"
02 17
03 XEQ 13
04 16
05 -
06 LASTX
07 /
08 14
09 +
10 "*i"
11 XROM "DC"
12 "⊢"
13 ASTO c
14 ΣREG 11
15 FIX 2
16 XROM "GE"

17♦LBL "RF"
18 ",x̄*"
19 ASTO d
20 CF 03
21 CLA
22 RTN

23♦LBL "Σ?"
24 CLA
25 XROM "C?"
26 RCL \
27 XEQ 14
28 CLA
29 X<>Y
30 -
31 RTN

32♦LBL "S?"
33 XROM "C?"
34 CHS

35♦LBL 13
36 64
37 MOD
38 SF 25

39♦LBL 02
40 RCL IND X
41 FC? 25
42 RTN
43 X<> L
44 +
45 GTO 02

46♦LBL "C?"
47 RCL c

48♦LBL 14
49 STO [
50 "⊢♦♦A"
51 X<> [
52 X<> d
53 CF 01
54 CF 02
55 CF 04
56 CF 07
57 FS?C 10
58 SF 07
59 FS?C 11
60 SF 09
61 FS?C 12
62 SF 10
63 FS?C 13
64 SF 11
65 FS?C 14
66 SF 13
67 FS?C 15
68 SF 14
69 FS?C 16
70 SF 15
71 X<> d
72  E38
73 /
74 INT
75 DEC
76 RTN

77♦LBL "DT"
78 "♦♦8♦!
      ."
79 RCL [
80 ",,,,,,"
81 ASTO L
82 ARCL L
83 AON
84 PSE
85 AOFF
86 ":::"
87 ASTO L
88 ARCL L
89 ARCL L
90 ARCL L
91 X<> d
92 AVIEW
93 STOP
94 X<> d
95 RDN
96 CLD
97 RTN

98♦LBL "AD"
99 RCL ↑
100 RCL ]
101 .
102 X<> \
103 "⊢******"
104 RCL \
105 CLA
106 STO [
107 ASTO X
108 RDN
109 STO [
110 RDN
111 STO \
112 RDN
113 STO ]
114 .1
115 STO ↑
116 ASHF
117 ARCL Z
118 RTN

119♦LBL "XE"
120 XEQ 14

121♦LBL 14
122 "⊢**"
123 STO \
124 RDN
125 SF 14
126 RCL b
127 X<> \
128 FC? 14
129 CLA
130 FC?C 14
131 RTN
132 "⊢***"
133 X<> [
134 X<> \
135 X<> [
136 "⊢**"
137 STO \
138 X<> ]
139 X<> \
140 "⊢**"
141 X<> \
142 STO b

143♦LBL "HD"
144 SIGN
145 RDN
146 RCL c
147 "8"
148 X<> [
149 "⊢**"
150 STO \
151 RDN
152 ASTO IND L
153 ΣREG IND L

154♦LBL "ΣC"
155 CLA
156 RCL c
157 STO ]
158 STO [
159 "⊢A"
160 CLX
161 STO \
162 "⊢B"
163 STO [
164 "⊢CD"
165 X<> ]
166 X<> d
167 SF 08
168 X<> d
169 X<> \
170 "⊢E"
171 X<> d
172 SCI IND \
173 X<> d
174 STO ]
175 RDN
176 RCL c
177 ΣREG 00
178 X<> c
179 STO [
180 "⊢F"
181 RDN
182 RCL \
183 "⊢GHI"
184 STO L
185 RDN
186 RCL c
187 STO [
188 X<> ]
189 "⊢J"
190 STO [
191 "⊢K"
192 X<> L
193 STO [
194 "⊢L"
195 X<> \
196 X<> c
197 RDN
198 CLA
199 END

01♦LBL 00
02 STOP
03 GTO "++"

04♦LBL "LB"
05 FS? 50
06 GTO 00
07 "DEC/HEX INPT"
08 XROM "VA"
09 CF 08
10 GTO 13

11♦LBL "L-"
12 CLA
13 XROM "VA"
14 CF 08
15 RCL a
16 STO [
17 RCL b
18 FS? 08
19 GTO 14
20 STO \
21 SF 08

22♦LBL 13
23 12
24 XROM "VS"
25 FC?C 25
26 PROMPT
27 CLST
28 STO 06
29 SIGN
30 ENTER↑
31 ENTER↑
32 R↑
33 GTO "++"

34♦LBL 00
35 RCL b
36 FC? 08
37 GTO 14
38 CLD
39 X<> [
40 STO a
41 X<> \
42 X<> b

43♦LBL 14
44 XROM "RT"
45 117
46 X<>Y
47 -
48 7
49 XROM "QR"
50 ST- Z
51 X<>Y
52 CHS
53 STO 09
54 X<> Z
55 LASTX
56 XROM "QR"
57 1.001
58 ST* 09
59 ST+ Y
60 FRC
61 ST* T
62 X<>Y
63 R↑
64 +
65 *
66 X<>Y
67 X<=0?
68 GTO 10
69 ST+ 09
70 7
71 *
72 +
73 STO 07
74 XROM "OM"
75 X<> c
76 STO 10
77 CLST

78♦LBL 06
79 STO 11
80 CLA

81♦LBL 07
82 ASTO 08
83 X<>Y
84 ISG 06

85♦LBL 15
86 SF 09
87 FS? 08
88 RTN
89 CF 22
90 CF 23
91 FIX 0
92 CF 29
93 "8"
94 ARCL 06
95 "⊢ OF "
96 ARCL 07
97 "⊢?"
98 XROM "VA"
99 TONE 7
100 STOP
101 FS? 48
102 GTO 14
103 FC? 22
104 GTO 19
105 GTO 08

106♦LBL 14
107 FC? 23
108 GTO 19
109 XROM "XD"

110♦LBL 08
111 X<0?
112 GTO 03
113 ENTER↑
114 CLA
115 ARCL 08
116 XROM "DC"
117 RCL 06
118 X<=0?
119 GTO 10
120 7
121 MOD
122 X≠0?
123 GTO 07
124 X<>Y
125 RCL 09
126 RCL 10
127 X<> c
128 RCL [
129 STO IND Z
130 X<>Y
131 X<> c
132 R↑
133 RCL 08
134 DSE 09
135 GTO 06
136 ISG 09

137♦LBL 20
138 CF 09
139 CLX
140 X<>Y
141 RCL 07
142 FRC
143  E3
144 *
145 AOFF
146 FIX 0
147 "SST, DEL 00"
148 ARCL X
149 FIX 3
150 XROM "VA"
151 BEEP
152 GTO 00

153♦LBL 01
154 RCL 06
155 X>0?
156 GTO 09

157♦LBL 10
158 "SST, MORE +'S"
159 XROM "VA"
160 TONE 3
161 GTO 00

162♦LBL 03
163 "*CORRECTION*"
164 XROM "VA"
165 TONE 6
166 FC? 09
167 GTO 01
168 DSE 06
169 GTO 14
170 ISG 06
171 GTO 10

172♦LBL 14
173 RCL 06
174 7
175 MOD
176 X=0?
177 ISG 09
178 GTO 14
179 RCL 11
180 X=Y?
181 GTO 13
182 STO 08
183 RDN
184 STO 11
185 GTO 09

186♦LBL 13
187 TONE 4
188 6
189 ST- 06
190 R↑
191 GTO 15

192♦LBL 14
193 CLA
194 ARCL 08
195 ARCL 10
196 CLX
197 X<> \
198 STO [
199 ASTO 08

200♦LBL 09
201 RDN
202 GTO 15

203♦LBL "-B"
204 FC? 08
205 GTO 15
206 FS? 09
207 GTO 08

208♦LBL 19
209 RCL 06
210 X<=0?
211 GTO 10
212 CHS
213 ISG X
214 7
215 MOD
216 X=0?
217 GTO 14
218 CLA
219 ARCL 08

220♦LBL 11
221 "⊢♦"
222 DSE X
223 GTO 11
224 X<> [

225♦LBL 14
226 RCL 09
227 X<>Y
228 RCL 10
229 X<> c
230 X<>Y

231♦LBL 12
232 STO IND Z
233 CLX
234 DSE Z
235 GTO 12
236 RDN
237 X<> c
238 RDN
239 GTO 20

240♦LBL "XD"
241 "⊢♦Δ"
242 RCL [
243  E2
244 XROM "QR"
245 29
246 ST- Z
247 -
248 .9
249 ST* Z
250 *
251 INT
252 X<>Y
253 INT
254 16
255 *
256 +
257 END

01♦LBL "VM"
02 CF 21
03 XROM "MT"
04 X<> [
```

APPENDIX J CONTINUED ON PAGE 349.

NP - NEXT PRIME

This routine will search to find prime factors of an integer n. More specifically, the routine begins its search from a starting trial divisor that the user inputs. NP returns only the next divisor of n. When NP is iterated on itself, starting with 2 as the first trial divisor, all the prime factors of n can be found one by one in increasing order. Intermediate processing can be done between successive prime factors and the routine NP can easily be returned to in order to continue the factorization of n. NP is valid for 10-digit integers n.

Example 1: Find the prime factors of 27,930.

The starting trial divisor will be 2.

Do:	See:	Result:
27930 ENTER↑ 2		enter initial inputs
XEQ " NP "	2	first prime factor
R/S	3	second prime factor
R/S	5	third prime factor
R/S	7	fourth prime factor
R/S	7	fifth prime factor
R/S	19	sixth prime factor
R/S	1	routine finished

The factor just before 1 is returned is the last prime factor. For this example,

27,930 = 2*3*5*7*7*19

Example 2: The number 40,013,933 is known to have only two prime factors, one of which is greater than 5000. Find the two factors of 40,013,933.

We may start with the next odd number greater than 5000.

Do:	See:	Result:
40013933 ENTER↑ 5001		Enter initial inputs
XEQ " NP "	5,309	after 41 seconds
R/S	7,537	second factor
R/S	1	routine finished

The two factors of 40,013,933 are 5,309 and 7,537.

COMPLETE INSTRUCTIONS FOR NP

NP will find the next divisor of an integer n starting from a given trial divisor d which may be 2 or any odd number. The search does not extend beyond the square root of n and if no divisor is found up to that point then n is returned. The divisor the routine returns will be prime provided n has no prime factors strictly smaller than d. Otherwise the divisor returned need not be prime. n may be any 10-digit integer.

1) The integer n may be any positive integer greater than or equal to 1. The trial divisor d must be 2 or an odd integer greater than 2.

2) Key n ENTER↑ d and XEQ " NP ".

3) The routine ends with n in Y and p in X where p is a divisor of n. p is also returned in LAST X.

4) If NP is executed from the keyboard, when the next divisor is returned, immediately pressing R/S will cause NP to continue searching for the next factor. The divisor returned may repeat, but when the routine returns 1 there are no more factors of n.

Example 3: Determine whether or not 99,991 is prime.

NP can be used to test potential primes by choosing 2 as the starting trial divisor. If the original number is returned then that number is prime. Key 99991 ENTER↑ 2 and XEQ " NP ". 99,991 is returned after about 41 seconds and hence 99,991 is prime.

MORE EXAMPLES OF NP

Example 4: Find all the prime factors of 4,019,788,151.

The starting trial divisor will be 2.

Do:	See:	Result:
4019788151 ENTER↑ 2		enter initial inputs
XEQ " NP "	37	first prime factor
R/S	89	second prime factor
R/S	163	third prime factor
R/S	7,489	fourth prime factor
R/S	1	routine finished

4,019,788,151 = 37*89*163*7489.

APPLICATION PROGRAM 1 FOR NP

The following routine called PNG for Prime Number Generator makes use of NP to generate prime numbers. Input to this routine is an odd number which serves as the starting point for the search for primes. If a printer is connected the generated primes will be printed.

```
LBL*PNG
2
LBL 01
XROM NP
X=Y?
VIEW X
CLX
2
ST + Y
GTO 01
```

Key in any odd number and XEQ "PNG". The following list of primes was obtained by keying in 3 and XEQ "PNG". Press R/S to end the routine when you are tired of looking at prime numbers.

LIST OF PRIMES

3	101	229	373	521	673	839
5	103	233	379	523	677	853
7	107	239	383	541	683	857
11	109	241	389	547	691	859
13	113	251	397	557	701	863
17	127	257	401	563	709	877
19	131	263	409	569	719	881
23	137	269	419	571	727	883
29	139	271	421	577	733	887
31	149	277	431	587	739	907
37	151	281	433	593	743	911
41	157	283	439	599	751	919
43	163	293	443	601	757	929
47	167	307	449	607	761	937
53	173	311	457	613	769	941
59	179	313	461	617	773	947
61	181	317	463	619	787	953
67	191	331	467	631	797	967
71	193	337	479	641	809	971
73	197	347	487	643	811	977
79	199	349	491	647	821	983
83	211	353	499	653	823	991
89	223	359	503	659	827	997
97	227	367	509	661	829	1009

LIST OF LARGE PRIMES

9,999,999,967	9,999,999,673
9,999,999,943	9,999,999,661
9,999,999,929	9,999,999,631
9,999,999,881	9,999,999,619
9,999,999,851	9,999,999,557
9,999,999,833	9,999,999,511
9,999,999,817	9,999,999,491
9,999,999,787	9,999,999,479
9,999,999,781	9,999,999,379
9,999,999,769	9,999,999,371
9,999,999,727	9,999,999,367
9,999,999,707	9,999,999,337
9,999,999,703	9,999,999,319
9,999,999,701	9,999,999,253
9,999,999,679	9,999,999,241

List of large primes found by Richard Nelson (1) using calls to NP.

APPLICATION PROGRAM 2 FOR NP

Use NP to help evaluate the Euler Phi-function $\phi(n)$, the number of integers smaller than and relatively prime to n. Two integers are called relatively prime if there is no prime number which is a common factor of both integers. An equivalent mathematical description is that the greatest common divisor of the two integers is 1.

The ϕ function is useful in the arithmetic of residues modulo an integer n, or in the structures of cyclic groups of n elements. For example, if an integer m is relatively prime to n, it is invertible modulo n and is a generator of the cyclic group of n elements. $\phi(n)$ is also the number of invertible residues mod n, or the number of generators of a cyclic group of n elements.

A closed form for ϕ is given by:

$\phi(0) = 0$ by convention
$\phi(1) = 1$ by convention
$\phi(p^k) = p^{k-1}(p-1)$ if p is prime
$\phi(m*n) = \phi(m)*\phi(n)$ if m & n relatively prime

The program "PHN" given here will determine $\phi(n)$ where n is the absolute value of the integral part of the number found in the X-register. In addition to the stack, it uses two extra registers M and N (these alpha registers may be replaced by two ordinary registers if desired). Register M contains the accumulation of a product which eventually builds up to $\phi(n)$. Register N carries successive prime factors of n. If a prime factor repeats, it is immediately multiplied to the product in the M register and the factorization continues via NP ; if the factor is new, the factor decreased by 1 is multiplied to the quantity in the M register. This is accomplished by a DSE X, which also detects the end of the factorization of n.

BAR CODE ON PAGE 480

APPLICATION PROGRAM FOR: NP

```
01♦LBL "PHN"        15♦LBL 03
02♦LBL C            16 ST* [
03 INT              17 R↑
04 ABS              18 R↑
05 X=0?             19 XROM "NP"
06 RTN              20 ST/ Y
07 E                21 ENTER↑
08 X=Y?             22 X<> \
09 RTN              23 RCL Y
10 STO [            24 X≠Y?
11 ENTER↑           25 DSE X
12 STO \            26 GTO 03
13 ENTER↑           27 RCL [
14 ST+ Z            28 RTN
```

Examples:
$\phi(2) = 1$
$\phi(17) = 16$
$\phi(41) = 40$
$\phi(697) = \phi(17*41) = 16*40 = 640$
$\phi(289) = \phi(17*17) = 17*16 = 272$

Routine Listing For: NP

```
98♦LBL e            112 X<> L
99♦LBL "NP"         113 2
100 RCL Y           114 X=Y?
101 SQRT            115 SIGN
102 LASTX           116 +
103 X<> Z           117 GTO 09
104♦LBL 09          118♦LBL 10
105 X>Y?            119 R↑
106 R↑              120 LASTX
107 R↑              121 X>Y?
108 X<>Y            122 ENTER↑
109 MOD             123 RTN
110 X=0?            124 ST/ Y
111 GTO 10          125 GTO e
```

LINE BY LINE ANALYSIS OF NP

Lines 98-103 set up the stack at LBL 09 as:
X: d Y: SQRT(n) Z: n T: n
where d=trial divisor and n=original integer

Lines 104-117 are the main loop in the program. The routine ends when the trial divisor d is greater than the square root of n, or when d is found to divide evenly into n. The increment in the trial divisor is 2 so that only odd divisors are used, except for the first time when d=2 the increment of 1 is obtained by the SIGN function at line 115.

Lines 118-123 end the routine with the divisor d in X and LAST X and the number n in Y.

Lines 124-125 are provided so that in manual mode the user need only press R/S to start the routine searching for the next prime divisor.

REFERENCES FOR NP

1. John Kennedy (918) PPC Journal V7N3P6
2. Jim Horn (1402) PPC Journal "Finding Factors Faster" V5N3P7.
3. Jim Horn (1402) PPC CALCULATOR JOURNAL "Fastest Factor Finder" V8N5P19

CONTRIBUTORS HISTORY FOR NP

John Kennedy (918) wrote an original version of NP and pointed out areas needed for improvement, especially execution speed. Phi Trinh (6171) came through with a much improved version. It is possible to reduce execution time at the expense of a larger program, but Phi's contribution made an improvement in speed while maintaining essentially the same program size. Phi's program also extended the validity range to 10-digit integers. John and Phi both contributed to the documentation of NP.

FINAL REMARKS FOR NP

NP has the advantage of being a versatile routine which can be used to not only find all the prime factors of an integer, but can also be used to generate primes and to test a given number for primeness. NP is very short, but is still over 3 times slower than the fastest known factor finding program, so speed is still an area for improvement. NP could be even faster if it did not have to check as a special case the prime 2.

FURTHER ASSISTANCE ON NP

John Kennedy (918) phone: (213) 472-3110 evenings
Phi Trinh (6171) phone: (206) 523-0940

TECHNICAL DETAILS

XROM: 20, 14 | NP | SIZE: 000 minimum

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
R00: not used

R06: NP requires no
R07: data registers as
R08: all computations
R09: are carried out
R10: in the stack
R11:
R12:

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode:
not used
FIX 0 recommended

Angular Mode:
not used

Unused Subroutine Levels:
5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine:
e, 09, 10

Execution Time: worst case time approximately $(0.129)*\sqrt{n}$ seconds for large primes n.

Peripherals Required: none

Interruptible? yes
Execute Anytime? no
Program File: FR
Bytes In RAM: 45
Registers To Copy: 36

Other Comments:

APPENDIX J - PPC CUSTOM ROM LISTING

APPENDIX J CONTINUED FROM PAGE 345.

```
05 RDN
06 VIEW ]
07 RCL d
08 FIX 9
09 VIEW [
10 STO d
11 RDN
12 LASTX
13 RTN

14♦LBL "EX"
15 CLA
16 X≠0?
17 "0Ω"
18 INT
19 X≠0?
20 CLA
21 RDN
22 LASTX
23 X<> [
24 ASHF
25 "⊢♦Δ"
26 ST- [
27 X<> [
28 RTN

29♦LBL "MT"
30 CLA
31 STO [
32 ASTO [
33 INT
34 X≠0?
35 "⊢♦"
36 X=0?
37 "⊢"
38 CLX
39 ST+ [
40 X<> [
41 RTN

42♦LBL "DS"
43 SIGN
44 RDN
45 RCL d
46 STO ]
47 SCI IND L
48 X<> d
49 STO [
50 "⊢*****"
51 X<> ]
52 STO [
53 "⊢**"
54 X<> \
55 STO d
56 RDN
57 CLA
58 RTN

59♦LBL "VS"
60 SF 25
61 INT
62 RDN
63 DSE T
64 ""
65 RCL IND T
66 RDN
67 FS? 25
68 RTN
69 "RESIZE>= "
70 TONE 3
71 R↑
72 RCL d
73 FIX 0
74 CF 29
75 ARCL L
76 STO d
77 RDN
78 RTN

79♦LBL "EP"
80 SF 25
81 XEQ "//"
82 FC?C 14
83 FC?C 25
84 GTO 14
85 XROM "PD"
86 3
87 -
88 7
89 /
90 INT
91 GTO 13

92♦LBL 14
93 XROM "C?"
94♦LBL 13
95 XROM "E?"
96 16
97 ST- Z
98 -
99 E3
100 /
101 +
102 XROM "OM"
103 .
104 DSE Z
105 XEQ 04
106 "♦-"
107 X<> [
108 STO IND Z
109 X<>Y
110 X<> c
111 R↑
112 XROM "GE"

113♦LBL 04
114 STO IND Z
115 DSE Z
116 GTO 04
117 RTN

118♦LBL "TN"
119 ""
120 XROM "DC"
121 "⊢"
122 ASTO T
123 SF 25
124 "Γ♦"
125 XROM "XE"
126 CF 25
127 RTN

128♦LBL "CX"
129 XROM "C?"
130 -

131♦LBL "CU"
132 ABS
133 RDN
134 RCL c
135 STO [
136 "⊢****"
137 11
138 X<> [
139 X<> d
140 STO ]

141♦LBL 00
142 RDN
143 X<> L
144 INT
145 X=0?
146 GTO 14
147 2
148 /
149 RCL [
150 X<>Y
151 FRC
152 X=0?
153 GTO 13

154♦LBL 01
155 FC?C IND Y
156 SF IND Y
157 FC? IND Y
158 CHS
159 X>0?
160 GTO 13
161 FC? IND Y
162 CHS
163 DSE Y
164 GTO 01

165♦LBL 13
166 DSE [
167 GTO 00

168♦LBL 14
169 X<> ]
170 X<> d
171 STO [
172 "⊢ABC"
173 X<> \
174 X<> c
175 RDN
176 CLA
177 RTN

178♦LBL "CD"
179 "⊢♦↓****"
180 RCL [
181 FS? 10
182 "⊢*"
183 STO [
184 CLX
185 X<> ]
186 SIGN
187 CLX
188 X<> \
189 "⊢***"
190 X<> [
191 X<> L
192 X<> \
193 INT
194 ST+ ]
195 RDN
196 6
197 ST* ]
198 RDN
199 E1
200 ST* L
201 X<> L
202 ST+ ]
203 CLX
204 X<> ]
205 END
[01♦LBL "VK"]
02 SF 21
03 FS? 55
04 PRKEYS
05 FS? 55
06 RTN
07 CF 21
08 8
09 RCL '
10 XEQ 07
11 "⊢0"
12 X<> [
13 X<> d
14 RCL e
15 XEQ 07
16 "⊢ "
17 X<> Z
18 X<> [

19♦LBL 01
20 -27.00008
21 RCL [
22 -
23 STO \
24 RDN

25♦LBL 02
26 FC? IND \
27 FC? 50
28 GTO 05
29 X<> d
30 FC? IND \
31 FC? 50
32 GTO 05
33 X<> d

34♦LBL 03
35 ISG \
36 GTO 02
37 DSE [
38 GTO 01
39 X<>Y

40♦LBL 04
41 STO d
42 CLST
43 CLA
44 PSE
45 CLD
46 RTN

47♦LBL 05
48 X<> d
49 35
50 RCL \
51 INT
52 +
53 OCT
54 1
55 ST+ Y
56 %
57 +
58 10
59 MOD
60 LASTX
61 *
62 INT
63 STO a
64 43
65 -
66 ABS
67 1
68 X<Y?
69 ST+ a
70 FS? 42
71 FC? IND \
72 CHS
73 ABS
74 X<> [
75 RDN
76 X<> \
77 RDN
78 " "
79 FC? 42
80 "    -"
81 FS? 50
82 X<> d
83 X<>Y
84 FC? 50
85 GTO 04
86 X<> d
87 X<> _
88 CLX
89 RCL d
90 FIX 0
91 CF 29
92 ARCL a
93 ISG L
94 GTO 06
95 "⊢ -"
96 ARCL a

97♦LBL 06
98 STO d
99 X<> _
100 AVIEW
101 TONE 0
102 R↑
103 STO \
104 R↑
105 STO [
106 RDN
107 RDN
108 X<> d
109 X<>Y
110 FS? 42
111 X<> d
112 GTO 03

113♦LBL 07
114 CLA
115 X<> [
116 "⊢*****"
117 X<> \
118 X<> [
119 RTN

120♦LBL "AL"
121 CLA
122 CF 10
123 XEQ 14
124 XEQ 14
125 X=Y?
126 XEQ 13
127 CLA
128 SF 10
129 X<=Y?
130 CF 10
131 FC?C 25
132 GTO 12
133 X<=Y?
134 RTN
135 X<> IND T
136 X<> IND Z
137 X<> IND T
138 RTN

139♦LBL 12
140 X<=Y?
141 GTO 12
142 R↑
143 X<> Z

144♦LBL 12
145 R↑
146 R↑
147 RTN

148♦LBL 13
149 R↑
150 R↑
151 SF 10
152 XEQ 14
153♦LBL 14
154 SF 25
155 ARCL IND Y
156 FC? 25
157 ARCL Y
158 "⊢*****"
159 ASTO L
160 ARCL L
161 "⊢*****"
162 .
163 FC? 10
164 GTO 14
165 STO \
166 "⊢**"

167♦LBL 14
168 STO [
169 "⊢**"
170 X<> \
171 RTN

172♦LBL "NC"
173 CF 25
174 GTO 14

175♦LBL "SU"
176 SF 25

177♦LBL 14
178 INT
179 E1
180 X<>Y
181 -
182 RCL d
183 SCI IND Y
184 ARCL Y
185 STO d
186 RDN
187 X<> ]
188 FS? 25
189 RCL ↑
190 "⊢*"
191 FC?C 25
192 GTO 14
193 X<> Z
194 STO ]
195 "⊢******"
196 X<> Z
197 STO ↑
198 RDN
199 X<> ]
200 X<> \
201 STO [
202 DSE L
203 CLX
204 X<> L
205 10↑X
206 RCL d
207 FIX 0
208 CF 29
209 ARCL Y
210 STO d
211 RDN
212 CLX
213 ISG L
214 CLX
215 X<> ↑
216 STO \
217 CLX
218 X<> ]
219 STO [
220 RDN
221 RTN

222♦LBL 14
223 X<> ]
224 CLA
225 STO [
226 ASTO X
227 END
[01♦LBL "NH"]
02 CLA
03 STO [
04 SIGN
05 X<> d
06 "⊢***"
07 .
08 X<> [
09 "⊢**"
10 X<> [
11 FIX 9
12 ARCL \
13 "⊢**"
14 ARCL ]
15 X<> ]
16 FIX 3
17 ARCL ]
18 STO ]
19 "⊢*"
20 RDN
21 STO d
22 CLX
23 FS? 10
24 GTO 12
25 RDN
26 14

27♦LBL 01
28 RCL ]
29 X<> d
30 FC? 06
31 FS? 05
32 FC? 04
33 GTO 14

34♦LBL 13
35 SF 01
36 CF 02
37 CF 03
38 CF 04
39 FS?C 07
40 GTO 14
41 SF 07
42 FS?C 06
43 GTO 14
44 SF 06
45 CF 05

46♦LBL 14
47 X<> d
48 STO [
49 "⊢*"
50 RDN
51 DSE X
52 GTO 01

53♦LBL 12
54 STO ↑
55 X<> ]
56 X<> \
57 STO [
58 X<> L
59 AON
60 RTN

61♦LBL "HN"
62 7
63 SIGN

64♦LBL 02
65 RDN
66 RCL \
67 X<> d
68 CF 11
69 CF 10
70 FC?C 09
71 GTO 14
72 SF 12
73 FC?C 15
74 SF 15
75 FS? 15
76 GTO 14
77 FC?C 14
78 SF 14
79 FC? 14
80 SF 13

81♦LBL 14
82 FS? 07
83 SF 11
84 FS? 06
85 SF 10
86 FS? 05
87 SF 09
88 FS? 04
89 SF 08
90 FC? 01
91 GTO 14
92 SF 08
93 FC?C 11
94 SF 11
95 FS? 11
96 GTO 14
97 FC?C 10
98 SF 10
99 FC? 10
100 SF 09

101♦LBL 14
102 X<> d
103 X<> ]
104 "⊢*"
105 STO ↑
106 "⊢*"
107 X<> ↑
108 STO ]
109 DSE L
110 GTO 02
111 CLA
112 STO [
113 AOFF
114 END
[01♦LBL "BL"]
02 2
03 STO [
04 X↑2
05 X↑2
06 X<>Y

07♦LBL 02
08 128
09 RCL [
10 ST+ [
11 /
12 XROM "QR"
13 RCL [
14 *
15 X<> Z
16 +
17 STOP
18 X<>Y
19 RDN
20 GTO 02

21♦LBL "FL"
22 CLA
23 CLST

24♦LBL 00
25 STOP
26 RCL X
27 8
28 XROM "QR"

29♦LBL 01
30 X<> [
31 X≠Y?
32 GTO 13
33 X<> [
34 7
35 X<>Y
36 -
37 2
38 X<>Y
39 Y↑X
40 ST+ \
41 E
42 X=Y?
43 XEQ 14
44 R↑
45 CHS
46 GTO 00

47♦LBL 13
48 X<> [
49 ENTER↑
50 XEQ 14
51 RDN
52 GTO 01

53♦LBL 14
54 CLX
55 X<> \
56 ISG [
57 ""
58 TONE 7
59 STOP
60 RTN

61♦LBL "BI"
62 -

63♦LBL 10
64 LASTX
65 +
66 STO IND Y
67 ISG Y
68 GTO 10
69 RTN

70♦LBL "IP"
71 XEQ 14
72 R↑
73 X<> c
74 RDN
75 RTN

76♦LBL "PS"
77 E3
78 /
79 +
80 ASTO Y
81 XEQ 14
82 FRC
83 E3
84 ST* Y
85 RDN
86 RCL d
87 FIX 0
88 CF 29
89 " "
90 ARCL L
91 "⊢ OFF;"
92 ARCL Y
93 "⊢ ON"
94 STO d
95 X<> L
96 INT
97 FC? 10
98 PROMPT
99 FC? 10
100 GTO 13
101 FS? 24
102 RTN
103 XROM "TN"
104 XROM "TN"
105 R↑
106 R↑

107♦LBL 13
108 240
109 RDN
110 CLX
111 X<> IND T
112 STO c
113 XEQ IND Z
114 XROM "GE"

115♦LBL 14
116 XROM "E?"
117 257
118 -
119 X<0?
120 GTO 14
121 R↑
122 SIGN
123 R↑
124 R↑
125 ΣREG T
126 XROM "OM"
127 STO [
128 240
129 ASTO IND X
130 R↑
131 R↑
132 RTN

133♦LBL 14
134 ENTER↑
135 XROM "S?"
136 +
137 "OVERSIZE"
138 XROM "VA"
139 XROM "GE"

140♦LBL "T1"
141 TONE 7
142 TONE 7
143 TONE 7
144 TONE 9
145 TONE 9
146 TONE 9
147 TONE 7
148 TONE 9
149 TONE 9
150 TONE 7
151 TONE 9
152 TONE 9
153 TONE 9
154 RTN

155♦LBL "MS"
156 4
157 ST+ 06
158 RCL IND 06
159 SIGN
160 DSE 06
161 RCL IND 06
162 DSE 06
163 RCL IND 06
164 DSE 06
165 RCL IND 06
166 DSE 06
167 RCL IND 06
168 END
[01♦LBL "IF"]
02 ABS
03 24
04 +
05 STO [
06 8
07 ST/ [
08 MOD
09 RCL d
10 X<> [
11 INT
12 SCI IND X
13 ARCL X
14 X<>Y
15 X<> ]
16 X<> \
17 X<> d
18 FC?C IND ]
19 SF IND ]
20 X<> d
21 STO [
22 RDN
23 12
24 -
25 SCI IND X
26 ARCL X
27 X<> ]
28 STO d
29 RDN
30 CLA
31 RTN
32 RTN

33♦LBL "CB"
34 X<>Y
35 XROM "PD"
36 X<>Y
37 XROM "PD"
38 -
39 RTN

40♦LBL "RT"
41 STO [
42 "⊢*****"
43 X<> \
44 XROM "2D"
45 2
46 /
47 INT
48 LASTX
49 FRC
50 512
51 GTO 14

52♦LBL "PD"
53 XROM "2D"
54 16
55 XROM "QR"
56 LASTX
57 X↑2

58♦LBL 14
59 *
60 RCL [
61 +
62 7
63 *
64 +
65 CLA
66 RTN

67♦LBL "DP"
68 7
69 XROM "QR"
70 X<>Y
71 16
72 ST* Z
73 X↑2
74 XROM "QR"
75 X<> Z
76 +
77 CLA
78 XROM "DC"
79 XROM "DC"
80 RCL [
81 RTN

82♦LBL "QR"
83 X<>Y
84 STO ]
85 X<>Y
86 MOD
87 ST- ]
88 LASTX
89 ST/ ]
90 CLX
91 X<> ]
92 X<>Y
93 RTN

94♦LBL "2D"
95 "*"
96 X<> [
97 X<> \
98 ASHF
99 "⊢♦↓♦'♦♦"
100 X<> [
101 X<> \
102 X<> [
103 "⊢♦♦B"
104 RCL [
105 INT
106 +
107 RCL \
108 *
109 ST+ [
110 X<> \
111 RCL ]
112 INT
113 HMS
114 *
115 RCL ]
116 +
117 E1
118 ST* [
119 *
120 X<> [
121 RTN

122♦LBL "SX"
123 XEQ 14
124 X<>Y
125 ENTER↑
126 X<> IND L
127 RDN
128 GTO 13

129♦LBL "RX"
130 XEQ 14
131 RCL IND L

132♦LBL 13
133 X<>Y
134 X<> c
135 RDN
136 RTN

137♦LBL 14
138 16
139 -
140 SIGN
141 RDN

142♦LBL "OM"
143 XEQ 14
144 "‡*i*"
145 X<> [
146 STO \
147 "⊢**"
148 X<> \
149 CLA
150 X<> c
151 RTN

152♦LBL "PA"
153 X<>Y
154 XROM "PD"
155 X<>Y
156 -
157 XROM "DP"

158♦LBL "GE"
159 SF 25
160 RCL b
161 CLA
162 FC?C 25
163 RTN
164 STO \
165 RDN
166 XEQ 14
167 X<> d
168 SF 05
169 SF 06
170 X<> d
171 STO [
```

APPENDIX J CONTINUED ON PAGE 367.

NR - NNN RECALL

NR is used to recall into the X-register an arbitrary seven byte (or shorter) hexadecimal code string previously stored in the numbered data registers through the use of PPC ROM routine **NS** (NNN Store).

Example 1: See the **NS** writeup for examples of the use of these routines.

COMPLETE INSTRUCTIONS FOR NR

Refer to the **NS** write-up for an explanation of the general purpose and nature of these routines. A Non-Normalized Number (NNN) or other hexadecimal code stored into registers R_{pqr} and R_{pqr+1} using **NS** is recalled into the X-register by placing the address pqr in the X-register and executing **NR**. More accurately, the address code that was used in executing **NS** to store the NNN must also be used to recall it. Thus, if the number 10.00002 was used to store the NNN into registers 10 and 12, 10.00002 must be placed in the X-register prior to executing **NR** in order to recall the NNN.

The NNN is recalled to X. Y, Z, T, and L remain unchanged. The register number that was in X is lost.

LINE BY LINE ANALYSIS OF NR

As is described in detail in the program description for **NS**, an arbitrary hexadecimal string bb_1, bb_2, bb_3, bb_4, bb_5, bb_6, bb_7, is stored by that routine into two numbered registers R_{mmm} and R_{nnn}, as alpha data, so that the contents of those registers are:

R_{mmm}: 10 2A bb_1 bb_2 bb_3 bb_4 bb_5

R_{nnn}: 10 00 00 00 00 bb_6 bb_7

Lines 15-17 of the routine append the contents of R_{mmm} to the previous contents of the alpha register, removing the alpha identifier byte 10 in the process, so that register M now contains: XX 2A bb_1 bb_2 bb_3 bb_4 bb_5 (the XX byte is the last byte of the previous contents of the alpha register). Lines 18-21 place that same code in the X-register, while the contents of register M are replaced by the contents of R_{nnn} (including the alpha identifier byte). Line 22 shifts that byte string leftwards five bytes into register N, leaving bb_6 and bb_7 as the first two bytes in M. Lines 23 and 24 complete the resynthesis of the NNN by overwriting the contents of register N with XX 2A bb_1 bb_2 ...bb_5 and shifting the alpha register leftward another two bytes. The original code is contained in register N, from where it is recalled by line 25.

CONTRIBUTORS HISTORY FOR NR

The first NNN storing and recalling routines (see *PPC CALCULATOR JOURNAL,* V8N4P16) were written by Bill Wickes (3735). Several intermediate versions were written by others as the state of the synthetic programming art advanced. This version of **NR** was written by Clifford Stern (4516).

FURTHER ASSISTANCE ON NR

Call Clifford Stern (4516) at (213) 748-0706.
Call Keith Kendall (5425) at (801) 967-8080.

Routine Listing For: **NR**

```
15♦LBL "NR"        21 X<> [
16 RDN             22 "├*****"
17 ARCL IND T      23 STO \
18 ISG T           24 "├**"
19 ""              25 X<> \
20 RCL IND T       26 RTN
```

TECHNICAL DETAILS

XROM: 20,50	NR	SIZE: 002

Stack Usage:		Flag Usage: NONE USED
0 T:	UNCHANGED	04:
1 Z:	UNCHANGED	05:
2 Y:	UNCHANGED	06:
3 X:	RECALLED CODE	07:
4 L:	UNCHANGED	08:

Alpha Register Usage:		
5 M:	ALL USED	09:
6 N:		10:
7 O:		
8 P:		25:

Other Status Registers: NONE USED
9 Q:
10 ├:
11 a:
12 b:
13 c:
14 d:
15 e:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

ΣREG: UNCHANGED

Data Registers: TWO CONSECUTIVE REGISTERS SPECIFIED BY THE USER.

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: .6 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	NS
Bytes In RAM:	32
Registers To Copy:	16

Other Comments:

NOTES A

NS - NNN STORE

NS will store any hexadecimal code string of up to seven bytes into a user-selected pair of numbered data registers. The code can thereafter be recalled into the X-register without normalization using PPC ROM routine NR (NNN Recall).

Example 1: Use NS and NR to store and retrieve when needed a particular configuration of flag register d, including any desired combination of display modes and user flag settings. As a general matter, the contents of register d cannot be recalled from numbered data registers without normalization. One way around this difficulty, is to set the first four flag bits to 0001 (flags 0-2 clear, 3 set) thus, ensuring that the flag register data will be treated as alpha data and therefore, will not be normalized when recalled. Although that procedure will often suffice, there may be instances in which the constraint imposed on the use of the first four user flags is objectionable. In such instances, NS and NR offer an alternative that places no constraints on the configuration of the flag register.

As an example, suppose that the user wishes to preserve the configuration USER, FIX 5, RAD, with flags 00, 01, 03, 06 and 07 set. Creating this configuration would normally require 35 keystrokes (manually) and 15 bytes in program (assuming that no flags need to be cleared), each time the configuration is needed. Using NS and NR the flag register information can be maintained in a numbered data register pair and recalled as desired. Have RCL d and STO d assigned to keys, and set the calculator to the flag configuration described above.

1.	RCL d	-3.00004	Contents of register d displayed in the X-register. Note the annunciators.
2.	0	0	Determines storage registers R00, R01.
3.	XEQ NS	-3.00004	Same as step 1.
4.	CLX; USER (off); XEQ "DEG"; FIX 4; clear flags 00, 01, 03, 06 and 07.	0.0000	Special configuration undone.
5.	XEQ NR	-3.0000	Original code reassembled in the X-register.
6.	STO d	-3.0004	Original flag status restored. Note the reappearance of the annunciators.

The particular flag setting chosen above will be normalized by a normal RCL. To see this, continue with:

7.	STO 02		
8.	RCL 02	<⊼⊠	The code in the X register has been normalized to alpha data.
9.	STO d		Note that the two flags 00 and 01 have been cleared.

The following sequence will interchange two different flag register configuration between register d and registers R00 - R01:

```
01 CLX          04 .
02 XEQ NR       05 XEQ NS
03 X<>d
```

COMPLETE INSTRUCTIONS FOR NS

A "Non-Normalized Number " (NNN) is a seven byte hexadecimal code string other than: (1) a string whose first nybble is "1" (alpha data), or (2) a code whose first and twelfth nybbles (mantissa sign and exponent sign) are "0" or "9"--the normal positive and negative sign codes--which contains no A, B, C, D, E, or F nybbles, and whose second nybble is not zero.

NNN's may be freely transferred between status registers using stack and synthetic operations, such as RCL Y and X<>M. However, the operations RCL pq, ARCL pq, X<>pq and VIEW pq, where Rpq is a numbered data register, generally will alter, or "normalize", the code of an NNN contained in the register. This normalization acts on both the recalled code and the register contents. The PPC ROM routines NS and NR provide a means to store NNN's into, and to recall them from, numbered data registers without normalization. Two registers are needed to store each NNN.

The input required for NS is (1) the desired NNN, contained in the Y-Register, and (2) an address code in the form pqr.000st, to identify the two storage registers. The address code, which follows the format used for the ISG function is contained in the X-Register. Storage into Registers R00 and R01 is accomplished as follows. The NNN is placed into the X-register with the stack lift enabled by keying in 0 and executing NS. The address code 2E-5, on the other hand will use R00 and R02 as the storage registers.

As is more fully explained in the analysis below, the first five bytes of the NNN are stored in the lower numbered register, while the last two bytes are stored in the higher numbered register. To a limited extent, therefore, fragments of different NNNs can be "spliced" together using these routines, if nonadjacent registers are used for storage.

Once stored using NS, the NNN can thereafter be recalled into the X-register using the procedures set forth in the program description for NR. These routines will work with any seven-byte hexadecimal code; their use is not limited to NNN's.

NS leaves the original code in register X. The previous contents of registers Z and T are preserved in X and Y, respectively. Lastx contains the original data register pointer after one ISG. T contains a NNN that can be used as zero.

Routine Listing For: NS	
01♦LBL "NS"	08 ASTO IND L
02 SIGN	09 ASHF
03 RDN	10 ISG L
04 ENTER↑	11 ""
05 "*"	12 ASTO IND L
06 X<> [	13 RDN
07 STO \	14 RTN

TECHNICAL DETAILS

XROM: 20,49 | **NS** | SIZE: 002

Stack Usage:
0 T: USED
1 Z: T
2 Y: Z
3 X: Y
4 L: X + 1

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Alpha Register Usage:
5 M:
6 N: ALL USED
7 O:
8 P:

Other Status Registers:
9 Q:
10 ⊢: NONE USED
11 a:
12 b:
13 c:
14 d:
15 e:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

ΣREG: UNCHANGED

Data Registers:
R00: TWO CONSECUTIVE REGISTERS SPECIFIED BY THE USER.
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: .5 seconds.

Peripherals Required: NONE

Interruptible? YES

Execute Anytime? NO

Program File: NS

Bytes In RAM: 25

Registers To Copy: 27

Other Comments:

LINE BY LINE ANALYSIS OF NS

Hex code can be recalled from numbered data registers without normalization if it is stored as alpha data. In normal operation, the HP-41C uses the functions ASTO and ARCL to transfer alpha data to and from the alpha register. These functions preserve data as alpha data by appending the alpha identifier byte "10" to the front of the code when it is transferred from the alpha register with ASTO. The identifier byte is removed when the code is transferred back with ARCL. Appending the identifier byte means, however, that a maximum of six bytes of the code can be stored in one register as alpha data. The routines NR and NS overcome this limitation by dividing the NNN to be stored into two segments of fewer then six bytes each, and storing each segment into a separate data register.

Lines 01-03 of the program place the address pqr of the lower storage register into register L, and copy the desired NNN = bb_1 bb_2 bb_3 bb_4 bb_5 bb_6 bb_7 into the X- and Y- registers. Line 05 places the byte 2A into the rightmost byte pointer of register M. Lines 06-07 manipulate that byte and the NNN to create, in the alpha registers, the string 2A bb_1 bb_2 ...bb_7. The function of the "*" byte 2A is to prevent suppression of any leading null bytes in the NNN.

Line 08 appends the alpha-identifier byte 10 onto the the front of the string and stores the first seven bytes of the string (as modified) into the register designated by register L. Thus, the code stored in that register is 10 2A bb_1 ...bb_5. Line 09 deletes the leftmost six bytes from the string in alpha. Lines 10-14 complete the routine by storing the remaining two bytes bb_6 and bb_7 into the second data register as alpha data.

CONTRIBUTORS HISTORY FOR NS

The first NNN storing and recalling routines (see *PPC CALCULATOR JOURNAL*, V8N4P16) were written by Bill Wickes (3735). Several intermediate versions were written by others as the state of the synthetic programming art advanced. This version of NS was written by Clifford Stern (4516).

FURTHER ASSISTANCE ON NS

Call Clifford Stern (4516) at (213) 748-0706.

Call Keith Kendall (5425) at (801) 967-8080.

NOTES

OM - OPEN MEMORY

OM lowers the "curtain" defining the beginning (R00) of the user data register block to 010_{16}. (See the memory map in the LF writeup.) This curtain location permits access to all of user memory (as R_{176} and up) while protecting the status registers from accidental normalization and preventing MEMORY LOST or loss of Catalog 1 on interruption. OM is used as a subroutine by all PPC ROM routines which alter program memory or key assignments.

Example 1: If F? doesn't agree to within one register of the number of free registers displayed, you have some miscellaneous data or garbage stored between the key assignments and the .END. of program memory. (See the memory map in the LF writeup.) An easy remedy to this situation is to execute CK. But this clears key assignments along with the unwanted data. As long as there is at least one empty register above the key assignments the following program sequence (which can also be performed from the keyboard) will clear the unwanted data while leaving the key assignments intact.

```
01  LBL"BCF"  (Block Clear of
02  XROM LF    Free Registers)
03  X<>Y
04  X<>c
05  X<>Y
06  ISG X
07  XROM BC
08  X<>Y
09  X<>c
10  RTN
```

This probably should have been an application program for LF, but it should be noted that LF calls OM, leaving the result in Y on completion. In fact, lines 03 and 04 above can be replaced by XROM OM with the same result. OM is actually doing double duty in this example.

COMPLETE INSTRUCTIONS FOR OM

OM does not require any input. It may be executed manually or as a program routine. When execution is completed: (1) The "curtain" defining the beginning (R00) of the user data register block is lowered to the fictitious address 010_{16}, and (2) the statistics register block is relocated so that statistical operations are inoperative. The pointer to the permanent .END. is left unchanged. The original contents of status register c are preserved by OM in the X-register. Therefore, the operation STO c (or X<>c) can be used, subsequent to execution of the routine, to return the statistics register block and the "curtain" to their pre- OM locations. To ensure this result, care should be taken, during the course of calculator operations made while the "curtain" is lowered, not to lose the previous contents of register c. Attempting to recover from OM by resizing will cause MEMORY LOST. The safest recovery procedure is to XEQ CX with an input of 256 + n*64, where n is the number of single density memory modules present. If you know the previous SIZE, use 256 + n*64 - SIZE.

While the curtain is lowered by OM, all user memory registers can be accessed indirectly as data registers, starting with $0C0_{16}$ (bottom of memory) = R176. The status registers are not redesignated and may be accessed directly by "normal" synthetic instructions. Care should be taken that the permanent .END. is not lost inadvertently, which would disrupt the global label chain. By subtracting 16 from the address of the .END. register given by PPC ROM routine E?, its temporary address as a data register may be determined and, thus, avoided.

OM saves the original contents of X, Y, and Z in Y, Z, and T respectively. LastX is undisturbed, and ALPHA is cleared.

MORE EXAMPLES OF OM

Example 2: Use OM to clear a block of program registers. RCL b at the starting location and again at the ending location (as for CB). Then execute the following program:

```
LBL "BCP"      (Block Clear of Program Memory)
X<>Y
XROM PD
7
/
INT
17
-
E3
/
X<>Y
XROM PD
7
/
INT
15
-
+
XROM OM
X<>Y
XROM BC
X<>Y
X<>c
RDN
RTN
```

Avoid using "BCP" to clear alpha labels and ENDs, unless there will be an unpacked END remaining below the cleared section to restore Catalog 1 label linkage (by PACKing).

Routine Listing For: **OM**

```
142♦LBL "OM"        183♦LBL 14
143 XEQ 14          184 RCL c
144 "▮*i*"          185 STO [
145 X<> [           186 "├♦♦♦♦♦"
146 STO \           187 X<> [
147 "├**"           188 X<> d
148 X<> \           189 CF 00
149 CLA             190 CF 01
150 X<> c           191 CF 02
151 RTN             192 CF 03
                    193 X<> d
                    194 RTN
```

TECHNICAL DETAILS

XROM: 10,58	OM	SIZE: 000

Stack Usage:	Flag Usage: SEVERAL USED BUT ALL RESTORED
0 T: Z	04:
1 Z: Y	05:
2 Y: X	06:
3 X: old c	07:
4 L: UNCHANGED	08:
Alpha Register Usage:	09:
5 M:	10:
6 N: ALL CLEARED	
7 O:	
8 P:	25:

Other Status Registers:	Display Mode: UNCHANGED
9 Q: NOT USED	
10 ⊢: NOT USED	
11 a: NOT USED	Angular Mode: UNCHANGED
12 b: NOT USED	
13 c: ALTERED	
14 d: USED BUT RESTORED	Unused Subroutine Levels:
15 e: NOT USED	5

ΣREG: UNCHANGED	Global Labels Called:	
Data Registers: NONE USED	Direct	Secondary
R00:	PART OF GE	NONE
R06:		
R07:		
R08:		
R09:		
R10:		
R11:	Local Labels In This Routine:	
R12:	NONE	

Execution Time: .8 seconds.

Peripherals Required: NONE

Interruptible? YES	Other Comments:
Execute Anytime? YES	Be sure not to lose the old c register contents that end in X.
Program File: IF	
Bytes In RAM: 54	
Registers To Copy: 60	

LINE BY LINE ANALYSIS OF OM

Lines 142-143 use subroutine LBL 14 (lines 183-194 of the IF group) to place the hexadecimal digit string 0stu as the first two bytes in the X-register, where stu is the absolute address of the register containing the permanent .END.. Line 144 creates the code 1FF0016901 (in hexadecimal digits), which is combined with 0stu by lines 145-148 to assemble, in the X-register, the hexadecimal digit string 1FF00169010stu. This string, when placed into register c at line 150, leaves the .END. at its prior location in register stu, but relocates the statistical registers.

The principal purpose of the routine is to relocate the "curtain" by defining the first data register R00 as the nonexistent register at address 010_{16}. This is accomplished by placing the hexadecimal digits 010 into the 9th, 10th and 11th nybble positions of register c. 010 is the only address between the status registers and program memory that will not cause MEMORY LOST when the curtain is assigned to it. See the CX writeup for more information on this.

The leading three digits of register c define the first of the 6 statistical registers. Its relocation to address 1FF ensures inclusion of nonexistent addresses within the statistical register block (thus preventing execution of the statistical functions), and the leading nybble 1 of the code ensures that it will be treated as ALPHA data (i.e., not normalized) when recalled from a numbered data register.

REFERENCES FOR OM

Brief note in *PPC CALCULATOR JOURNAL*, V8N2P37a. OM was named CT16 and C16 in earlier ROM progress columns.

CONTRIBUTORS HISTORY FOR OM

OM was written by Roger Hill (4940), completely revising an earlier version by Keith Jarett (4360). The example applications were written by Keith Jarett.

FURTHER ASSISTANCE ON OM

Call Roger Hill (4940) at (618) 656-8825.

Call Keith Kendall (5425) at (801) 967-8080.

NOTES

PA - PROGRAM POINTER ADVANCE

PA is effectively a selectable byte jumper. It accepts a RAM program pointer in Y (usually obtained by RCL b) and a decimal number in X, creating a new program pointer that points x bytes below the original pointer in RAM program memory.

Example 1: Assign RCL b and STO b to any convenient keys using MK. The decimal codes are 144,124 and 145,124. XEQ GE and key in a TONE 5 instruction in PRGM mode. Go to RUN mode, RCL b using the key assignment, and press 1 XEQ PA. This produces a program pointer 1 byte down from the one you got by RCL b. Now press the STO b assigned key and go to PRGM mode. SST to see 01 LBL 04. This LBL 04 represents the postfix 5 for the TONE 5 instruction. (see the combined Byte Table). Let's change this postfix to create a synthetic tone. Backarrow the LBL 04 and insert STO 09 in its place. GTO .001 to cause the 41C to recompute the line number and put you back at the true line 01, which now shows TONE 7 (it's actually TONE 57--only the last decimal digit is shown for synthetic TONES). SST in RUN mode to hear the tone.

COMPLETE INSTRUCTIONS FOR PA

To use PA to advance the program pointer n bytes in RAM from any chosen location, perform these steps:

1) Assign RCL b and STO b to keys
2) RCL b (in RUN mode) at the chosen location.
3) Put n in the X register
4) XEQ PA
5) STO b

You can then switch to PRGM mode and edit the following bytes as desired. PA cannot be used as a subroutine because it ends with GE. This is for two reasons. First, the STO b following XEQ PA must be done with the program pointer in RAM (see the Ab writeup for reasons). GE gets the program pointer back from ROM to RAM when PA is called from the keyboard. Second, GE sets the line number to 00, which is a big help in editing the following bytes. This technique of setting the line number to zero was discovered by Roger Hill (4940) and is discussed further in section 5G of *SYNTHETIC PROGRAMMING ON THE HP-41C.*

PA clears the alpha register and uses the whole stack. However, the decimal increment, originally in x, winds up in y.

Because PA cannot be used in a subroutine, you'll have to use another approach to advance the pointer in a running program. This approach is described in Application Program 2 for DP.

MORE EXAMPLES OF PA

Example 2: Create the synthetic text line "⊦μ VOLTS". First XEQ GE and key in 01 "⊦ZVOLTS". Then in RUN mode push the RCL b assigned key, then 2, XEQ PA. When the result appears press STO b (key assignment)

TECHNICAL DETAILS

XROM: 10,59	**PA**	SIZE: 000

Stack Usage:
- 0 T: zero
- 1 Z: USED
- 2 Y: X
- 3 X: result
- 4 L: USED

Flag Usage: MANY USED BUT ALL RESTORED
04: 05: 06: 07: 08: 09: 10: 25:

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: CLEARED
- 12 b: BYTES 1-5 CLEARED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 0

ΣREG: UNCHANGED

Data Registers: NONE USED
R00: R06: R07: R08: R09: R10: R11: R12:

Global Labels Called:

Direct	Secondary
PD	2D
DP	QR
	DC

Local Labels In This Routine: NONE

Execution Time: 5.6 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	IF
Bytes In RAM:	13
Registers To Copy:	60

Other Comments:

and go into PRGM mode. You are now two bytes within the text line. (The append instruction is actually a character which takes up one byte of the text line. When the first character of a text line is ⊦, it is intrepreted as a control character indicating the append operation.) SST and backarrow the COS instruction, which corresponds to the character Z. Key in LBL 11 to replace it. LBL 11 corresponds to the synthetic character μ. GTO .001 to see the result. Editing operations like this can be done with PA anywhere in user program memory. PA can also be used to view postfix bytes of any instruction.

LINE BY LINE ANALYSIS OF PA

Line 154 converts the program pointer to a decimal number (of bytes). The number of bytes to be advanced is subtracted from this decimal number, then line 154 converts the result back to a program pointer. Line 158 begins GE which causes the program to halt at line 00 of the last user program in RAM.

REFERENCES FOR PA

See *SYNTHETIC PROGRAMMING* by Bill Wickes (3735), section 5G for a discussion of enhanced byte jumping. See *PPC TECHNICAL NOTES*, V1N5P60 by Bill Wickes for a presentation of a programmable byte jumper analogous to PA.

CONTRIBUTORS HISTORY FOR PA

PA is a simple application of the DP and PD routines. Roger Hill (4940) made the most important contribution by suggesting the use of GE at the end of PA.

FURTHER ASSISTANCE ON PA

Call Roger Hill (4940) at (618) 656-8825.

Call Keith Kendall (5425) at (801) 967-8080.

Routine Listing For: **PA**

```
152♦LBL "PA"
153 X<>Y
154 XROM "PD"
155 X<>Y
156 -
157 XROM "DP"

158♦LBL "GE"
159 SF 25
160 RCL b
161 CLA
162 FC?C 25
163 RTN
164 STO \
165 RDN
166 XEQ 14
167 X<> d
168 SF 05
169 SF 06
170 X<> d
171 STO [
172 CLX
173 X<> \
174 "⊦**"
175 STO [
176 "⊦**"
177 X<> \
178 CLA
179 X<> [
180 RDN

181♦LBL "Ab"
182 ASTO b

183♦LBL 14
184 RCL c
185 STO [
186 "⊦♦♦♦♦♦"
187 X<> [
188 X<> d
189 CF 00
190 CF 01
191 CF 02
192 CF 03
193 X<> d
194 RTN
```

NOTES

PD - PROGRAM POINTER TO DECIMAL

PD decodes a RAM program pointer into a decimal number representing the number of bytes from the bottom of memory (status register T). PD is used in CB to convert two program pointers to decimal in order that the difference, or number of bytes between the two pointers, can be determined. PD is the inverse of DP (decimal to program pointer).

Example 1: Determine the number of bytes from the bottom of memory to the curtain. Set SIZE 017 for this example, and get to the top of program memory by executing CAT 1, pressing R/S immediately, and RTN to get to line 00. Then RCL b using an assigned key and XEQ PD. You'll see 1673, 2121, 2569, 3017, or 3465, depending on whether you have 0, 1, 2, 3, or 4 memory modules present. Note that 1673 = (256-017)*7. The top of mainframe memory is 256_{10}, the curtain is 17 registers down from there, and there are 7 bytes per register.

COMPLETE INSTRUCTIONS FOR PD

Go to any point in RAM, then RCL b using an assigned key, and XEQ PD. The decimal number of bytes corresponding to the program pointer will be returned in X. The contents of Y are saved, but the rest of the stack and the alpha register are lost.

Example 2: Check a synthesized pointer. Key in 1792, XEQ DP, XEQ PD. You should see 1792, since PD is the inverse of DP.

APPLICATION PROGRAM 1 FOR PD

PD converts a RAM pointer to decimal, but it will not givea meaningful result for a ROM pointer. This is due to the RAM vs. ROM program pointer differences discussed in the Ab write-up. But suppose you want to convert a ROM pointer to decimal, as part of a ROM byte counter (see CB Application Program 1) or for any other purpose. The following routine will do the job.

```
LBL "RPD" (ROM pointer to decimal)
XROM 2D
 256
  *
RCL M
  +
RTN
```

This routine is very similar to PD. The differences are a consequence of the pointer differences disscussed in the Ab write-up. If you understand PD and this routine, you understand the RAM and ROM pointer formats.

Routine Listing For: PD

52♦LBL "PD"	59 *
53 XROM "2D"	60 RCL [
54 16	61 +
55 XROM "QR"	62 7
56 LASTX	63 *
57 X↑2	64 +
	65 CLA
58♦LBL 14	66 RTN

TECHNICAL DETAILS

XROM: 10,52 — **PD** — SIZE: 000

Stack Usage:
- 0 T: Y
- 1 Z: Y
- 2 Y: Y
- 3 X: result
- 4 L: 7* reg. number

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

ALL CLEARED

Other Status Registers:
- 9 Q:
- 10 ⊢:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

NONE USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 5

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Global Labels Called:
Direct: 2D, QR
Secondary: NONE

Local Labels In This Routine: 14

Execution Time: 1.8 seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: IF
Bytes In RAM: 24
Registers To Copy: 60

Other Comments:

PD

LINE BY LINE ANALYSIS OF PD

RAM program pointers consist of two bytes of the form na bc_{16}, where $0 \leq n \leq 6$ is a byte number within the register abc_{16}.

Line 53 converts the last two bytes of x (the program pointer bytes) to decimal. The decimal translation of the last byte is left in M, that of the penultimate byte in X. See 2D for details. Lines 54 and 55 split the penultimate byte into two hexadecimal digits. The second digit is multiplied by 256 and added to the contents of M to give the decimal absolute address of the register addressed by the pointer. This is multiplied by 7 (lines 62 and 63) to get a number of bytes, which is then added to the first hexadecimal digit to get a total number of bytes.

CONTRIBUTORS HISTORY FOR PD

The first version of PD (see *PPC CALCULATOR JOURNAL*, V7N3P7) was written by William C. Wickes (3735). The ROM version was written by Roger Hill (4940) as part of a byte-saving program pointer manipulation package.

FURTHER ASSISTANCE ON PD

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

NOTES

PK - PACK KEY ASSIGNMENT REGISTERS

This routine performs a somewhat obscure, but useful function lacking in the HP-41's mainframe: it packs the function assignments in the key assignment registers, eliminating any "voids" created by former assignments having been deleted, so that the most efficient use is made of the assignment registers.

When a program line is deleted from program memory, it is replaced by nulls which are later eliminated when program memory is packed (either by the PACK or GTO.. instructions or automatically when program memory is filled). When a function key assignment is deleted, a similar gap or "void" is left in the key-assignment register (actually only the keycode is changed to zero, the other two bytes being unaltered--see part A of "Background for MK"), but this void does not get eliminated, even during packing, unless two voids accumulate in a register, in which case turning the machine off and on or packing will cause that register to be eliminated and the gap closed by moving down the key assignments above it. At no time does the HP-41 do any "horizontal" shifting of key assignments within the key assignment registers--unless we program it to do so by synthetic means. Routine PK is designed to do just that, combining two half void registers whenever they occur to make a full register, and closing all of the gaps left by voids in the assignment registers.

Example 1: XEQ CK to clear your function key assignments. Then ASN -, +, *, and / (in that order) to any 4 keys. XEQ A? to see that 2 assignment registers are being used. Now delete the assignments of + and / using ASN alpha alpha [key]. XEQ A? and you'll see that 2 assignment registers are still occupied. XEQ PK to pack the 2 remaining key assignments into one register. (The number of registers used is displayed on completion of PK.)

COMPLETE INSTRUCTIONS FOR PK

Simply execute PK if you feel that there is a need for the key assignments to be packed. It is not necessary (and usually not worth the time spent) to keep the assignments packed all the time, but there are times when there is a definite advantage in having them packed, for example (1) if more room in program memory is needed, and (2) prior to writing a status card containing key assignments. In the second case, packing the assignments before writing the card will not only make the most efficient use of the cards, but will also minimize the amount of free registers needed when the card is read back later.

Execution of PK will result in a decrease of the number of key-assignment registers (and a corresponding increase in the number of free registers) whenever there are at least two "void" half-registers. Such a situation might occur (1) when one or more key assignments are deleted, or (2) if an odd number of key assignments have been made using the normal ASN function and another odd number of assignments made using MK. If you are in doubt as to whether key-assignment packing is needed, executing PK will do no harm; if packing was not needed there may be at most a reordering of the key assignments, which is of no practical significance in normal situations. (If PK is executed twice, the second execution will not even change the ordering.)

When PK is executed, the assignment registers are recalled one by one until the .END. is reached, whichever comes first. If both halves of a register are non-void, then the assignment pair is re-stored in the lowest available register--after restoring the initial F0 byte which got changed to 10 in the recalling (see analysis of LF). If one half of a recalled register is void, then the non-void half is set aside until another half-void register is found, at which time the two non-void halves are combined, the former being the left half and the latter being the right half of a new assignment pair, which is them stored in the lowest available register. If there is a set-aside half register remaining at the end, it is stored as the left half-register above the rest of the assignments, the right half having three null bytes. This arrangement is compatible with MK and 1K, as subsequent use of either of these will create more assignments filling the null half-registers and extending on upward without creating any voids in-between.

After PK as been executed, the results (aside from the packing) are as follows:

T = Z = c-register contents for curtain at 16 (used during the program).

Y = bbb.eee showing the beginning (bbb) and end(eee) registers (both relative to a curtain address of 16) of the free-register block, including the half-null assignment register if any.

X = Number of assignment registers, to the nearest .5 (doubling this will give the number of function assignments).

L = .5 if there is a half-null register after packing, 1 if there is not.

Alpha = cleared.

Flags 09, 10 cleared.

WARNING: (1) PK may be temporarily interrupted, but do not abandon it in the middle without letting it terminate, as the curtain is lowered and the key assignments are temporarily disrupted during the program. (2) Flag 25 may be left set after PK has been executed--something to remember if you execute any instruction from the keyboard afterward that might normally give a "NONEXISTENT" or other error message. (3) Because there is backward branching with the curtain down this program must be preceeded by at least one END if copied into user program memory and executed there.

MORE EXAMPLES OF PK

See Figure 3, diagrams (d) and (e) of part A of "Background for MK" for an example of key assignments before and after the execution of PK. After execution, X will contain 3.5 (the number of assignment registers occupied) and Y will contain 179.eee, where eee is the absolute address of the .END. minus 17 (decimal). The number of free registers in the 00 REG... or .END. REG... display will increase by one as a result of executing PK.

LINE BY LINE ANALYSIS OF PK

For an understanding of the structure of key assignment registers and the terminology used here, it will be helpful to read Part A, "The Storage of Key Assignments" in Background for MK.

PK

Routine PK begins by calculating the absolute address of the final .END. (line 79), and after line 90 we have 176.eee in X and 175.eee in Y, where eee is the address of the register just below the .END. relative to a (decimal) curtain address of 16. Flags 09 and 10 have also been cleared. The curtain is lowered to 16 in line 91, and by the time line 93 is reached, the stack consists of T = 175.eee, Z = 176.eee, Y = old c-register contents, X = 0. Line 93 transfers us into the main loop in which the key assignment registers are recalled, packed, and stored. During virtually all of this process, the function of the stack and flags are as follows:

T = index for recalling the unpacked assignments
Z = index for storing the packed assignments
Y = old c-register contents
X = working register
L = half-register of assignments waiting to be combined with another half-register

Flag 09: When set, indicates that the left half of the register just recalled is "void".
Flag 10: When set, indicates that there is an assignment froma a previously recalled half-void register waiting in L.

The loop is entered at LBL 03 (line 148), where the recall-index T is changed to 176.eee and we go to LBL 01 to begin the recalling process (unless eee = 175, in which case there was no room for any key assignments and no looping is done). After LBL 01 (line 114) we get an assignment register storing zero in its place (line 115). Lines 116 through 119 send us to LBL 04 to terminate the packing process if the recalled register was empty. (Note: A simple X = 0? GTO 04 is not sufficient here, because X, if non-zero, will be alpha data for which X = 0? would cause an error message. In the present program if X = 0, then line (118 will be skipped and we go to LBL 04, while if X ≠ 0 then it will be alpha data and since flag 25 was set, the conditional will be ignored and flag 25 cleared, causing line 118 to test false and line 119, GTO 04, to be skipped). If the assignment register was not empty, then it (now in X) will be of the form 10 a_1 a_2 a_3 b_1 b_2 b_3, where the first byte 10 was originally F0 in the assignment register, but got normalized to 10 during the recalling (see the line-by-line analysis of LF for discussion of this), a_1 a_2 a_3 are the three bytes of the assignment in the "left half -register", and b_1 b_2 b_3 are the three bytes of the assignment in the "right half-register." Our task is now to check whether a_3 or b_3 (or both) is zero, indicating a "void" half-register.

Lines 120 and 121 simply put ten characters in the alpha register, and after line 124 we have N = 00 00 00 00 00 00 a_3, M = b_1 b_2 b_3 2A 2A 2A 2A (2A being the asterisk), the rest of the alpha registers being empty. In lines 125 through 127 we set flag 09 if a_3 = 00, indicating a void left half-register. After lines 128 through 133 we have only the single character b_3 in alpha (i.e., M = 00 00 00 00 00 00 b_3, N, O, and P being empty), while X contains 10 a_1 a_2 a_3 b_1 b_2 b_3 if a_3 ≠ 0, or b_1 b_2 b_3 2A 2A 2A 2A if a_3 = 00. In line 134 X and M are interchanged, and in the next two lines the byte F0 is appended to alpha if a_3 ≠ 00, in preparation for restoring the F0 byte of the key-assignment register. In lines 137 through 139 we branch to LBL 07 if flag 09 is set (i.e., a_3 = 00) or if b_3 = 00--in other words, if either half register is void, however, we continue with lines 140 through 143, which result in N = F0 a_1 a_2 a_3 b_1 b_2 b_3, the other alpha registers being irrelevant. (Line 143 simply effects a 6-character shift.) This reconstructed key-assignment pair is stored in the key assignment registers (lines 144 through 146), both the store index (in Z) and the recall-index (in T) are incremented (lines 147 through 149), and the loop repeated (line 150)--unless the .END. has been reached, in which case we go on to the termination sequence starting with LBL 04 (line 151).

Before discussing the termination sequence, however, consider what happens when one or both of the half-registers is void. In this case line 139 is executed, taking us to LBL 07 (line 94). Lines 95 through 98 cause a branch to LBL 03 (line 148) if both halves are void (that is, if Flag 09 was set and X = 0); note that line 97 if executed always causes a skip. In this case we simply increment the recall-index and start the loop over to fetch another assignment register. Otherwise, we go on to line 99 which results in X = a_1 a_2 a_3 b_1 b_2 b_3 F0 (if b_3 = 0), or b_1 b_2 b_3 2A 2A 2A 2A (if a_3 = 0). Note that in either case the first three bytes of X are the non-void half of the original assignment register; this half is to be packed with a similar half (if any) from some other assignment register. In lines 100 through 102 we go to LBL 08 (if flag 10 was set) to combine these three bytes with a half register previously stored in L, or else continue on (if flag 10 was clear) to store these three bytes in L for future combining. Flag 10 gets reversed in either case. Assuming flag 10 was originally clear, we create the bytes 2A 2A F0 in line 103, and the result of line 106 is X = 0, L = 10 2A 2A F0 a_1 a_2 a_3 (if b_3 = 0) or 10 2A 2A F0 b_1 b_2 b_3 (if a_3 = 0). Then in line 107 we rejoin the loop at LBL 03 to increment the recall-index and get a new assignment register. If, on the other hand, flag 10 was set before line 100, then we go to LBL 08 (line 108), where we put the previously saved half-register from L into alpha and tack on the three bytes from X, leaving (after line 112) an N register containing F0 followed by the two assignments just spliced together, all ready to be stored in the assignment register--which is done by branching (line 113) to LBL 02 to rejoin the loop at line 144, storing the assignments, and incrementing both indexes.

Finally, we consider the termination sequence starting with LBL 04 (line 151). This is reached either when line 149 causes a skip (corresponding to the .END. being reached by the recall-index) or when line 119 is executed (corresponding to an empty assignment register being recalled). If flag 10 is set there is an assignment waiting in L which has to be stored in the assignment registers after tacking on three null bytes for the right half-register (lines 154 through 157). Otherwise, we branch around those lines to LBL 05 (line 158). Lines 159 through 160 restore the curtain, and in lines 161 through 163 we put the last value of the store-index (as of when the loop was left) into X and Y and branch into the A? routine where the number of assignment registers is calculated--see A? for details.

TECHNICAL DETAILS

XROM: 10,09 | **PK** | SIZE: 000

Stack Usage:

0 T: TEMP. c FROM OM
1 Z: TEMP. c FROM OM
2 Y: bbb.eee à la LF
3 X: NUMBER OF ASSIGNMENT REGISTERS
4 L: USED

Alpha Register Usage:

5 M:
6 N: ALL CLEARED
7 O:
8 P:

Other Status Registers:

9 Q: NOT USED
10 ⊢: NOT USED
11 a: NOT USED
12 b: NOT USED
13 c: USED BUT RESTORED
14 d: USED BUT RESTORED
15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED

R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: ONLY FLAGS 9, 10, AND 25 ARE USED.

04:
05:
06:
07:
08:
09: CLEARED
10: CLEARED
25: USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
E?	2D
OM	PART OF GE
PART OF A?	

Local Labels In This Routine:

01 GET NEW REGISTER
02 STORE REGISTER
03 INCREMENT RECALL INDEX
04 TERMINATE
05
07 VOID HALF-REGISTER(S)
08
14 FORWARD BRANCHING

Execution Time:
18.3 seconds for 16 Assignment Registers.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	YES
Program File:	LF
Bytes In RAM:	168
Registers To Copy:	59

Other Comments:

Must be preceded by an END if copied into RAM.

Routine Listing For: PK

```
 78+LBL "PK"
 79 XROM "E?"
 80 17
 81 -
 82 E3
 83 /
 84 175
 85 +
 86 ENTER↑
 87 CF 09
 88 CF 10
 89 E
 90 +
 91 XROM "OM"
 92 .
 93 GTO 03

 94+LBL 07
 95 FS?C 09
 96 X≠0?
 97 FS? 09
 98 GTO 03
 99 X<> [
100 FS?C 10
101 GTO 08
102 SF 10
103 "**"
104 X<> [
105 X<> \
106 ASTO L
107 GTO 03

108+LBL 08
109 ARCL L
110 X<> [
111 STO \
112 "⊢***"
113 GTO 02

114+LBL 01
115 X<> IND T
116 SF 25
117 X≠0?
118 FS?C 25
119 GTO 04
120 "****"
121 ARCL c
122 STO [
123 "⊢****"
124 ASHF
125 X<> \
126 X=0?
127 SF 09
128 CLX
129 X<> \
130 FC? 09
131 STO [
132 X<> [
133 ASHF
134 X<> [
135 FC? 09
136 "⊢"
137 FC? 09
138 X=0?
139 GTO 07
140 RDN
141 RCL [
142 STO \
143 ARCL c

144+LBL 02
145 X<> \
146 X<> IND Z
147 ISG Z

148+LBL 03
149 ISG T
150 GTO 01

151+LBL 04
152 FC? 10
153 GTO 05
154 ARCL L
155 "⊢+++"
156 X<> [
157 STO IND Z

158+LBL 05
159 RDN
160 X<> c
161 X<>Y
162 ENTER↑
163 GTO 14
166+LBL 14
167 CLA
168 INT
169 175
170 -
171 .5
172 FC?C 10
173 SIGN
174 -
175 RTN
```

CONTRIBUTORS HISTORY FOR PK

The need for a key-assignment packing program was noticed in the summer of 1980 by Roger Hill (4940), who wrote a key-assignment packing program at that time and later modified it to make the present ROM program (whose improvements include executing slightly faster and not using any data registers). Bill Wickes (3735) also independently noticed the need for and wrote an assignment packing program around the same time.

Tom Cadwallader (3502) and Richard Collett (4523) suggested and wrote versions of synthetic key-assignment making programs that packed the assignments prior to prompting for the first new assignment, and the possibility was considered of using PK as a subroutine rather than LF to count the assignment registers in the MK program. Checking for voids and packing the assignment registers takes considerally more time than just counting them, however, and it was felt that in a typical situation where one or several assignments were to be made the occasional advantage of packing did not justify the extra waiting time involved.

It may be of interest to note a rather obscure bug which was found and corrected at the last minute: lines 121 and 143, ARCL c were originally ARCL Y (see the ROM listing mailed to orderers), which was correct in the original version, but occasionally caused errors after the program was modified for the ROM. In the ROM version, register c contains the lowered curtain address and also a Σ-register address of IFF (see OM), so that when ARCL'd it is treated as alpha data and always appends 6 characters to alpha. The erroneous ARCL Y caused the original c-register contents to be ARCL'd, which usually appended the desired 6 characters, but sometimes not, depending on the Σ-register absolute address and hence, on the data memory size and the ΣREG number. This erratic behavior showed up more when the SDS ROM simulator was loaded and used (because the user RAM was then free of long programs, allowing the Σ-register absolute address to be below 100 where the error was certain to manifest ifself), and for a while the SDS system was blamed for the problem. The cause was discovered after a rather lengthy cross-country phone conversation between the author and Richard Nelson (1), in which the routine was single-stepped through at both ends of the line, and the correction to the routine was able to be incorporated (along with a few other corrections in other routines--see PPC CJ, V8N2P32d) in a modified disk sent to HP. The present routine appears to be bug-free, although if he were to do it over again the author would include a CF 25 at the end to avoid possible confusion afterward (see Warning 2 of the instructions).

FURTHER ASSISTANCE ON PK

Call Tom Cadwallader (3502) at (406) 727-6869.
Call Roger Hill (4940) at (618) 656-8825.

NOTES

PM - PERMUTATIONS

This routine will compute the number of permutations of n objects taken k at a time. This number may be denoted by P(n,k) and may be described as the number of arrangements (orderings) of all subsets of size k selected from a set of n objects. More formally,

$$P(n,k) = n!/(n-k)! = n(n-1)(n-2)(n-3)\cdots(n-(k+1))$$

For PM the values n and k must satisfy the restriction 1<=k<=n.

Example 1: Compute P(10,5)

Key 10 ENTER↑ 5 and XEQ "PM". P(10,5) = 30240

COMPLETE INSTRUCTIONS FOR PM

1) To compute P(n,k) key n ENTER↑ k where 1<=k<=n.

2) XEQ "PM". The value P(n,k) will be returned in X. The value returned will not be exact if displayed in scientific notation. In this case however, the result displayed will be an accurate approximation.

The stack input/output for PM is as follows:

Input:		Output:	
T:	T	T:	n-k
Z:	Z	Z:	T
Y:	n	Y:	Z
X:	k	X:	P(n,k)
L:	L	L:	n-k

MORE EXAMPLES OF PM

Example 2: In a lottery there are 1st, 2nd, and 3rd prizes. If 100 tickets have been sold in how many ways can the prizes be awarded?

Key 100 ENTER↑ 3 and XEQ "PM". P(100,3) = 970200.

Example 3: The PPC Clubhouse conference table has room for 25 people. If 30 people arrive on a Friday night stuffing party, how many arrangements of members are available at the table?

Key 30 ENTER↑ 25 and XEQ "PM".

P(30,25) = 2.210440497 x 10^{30}. Note that in this problem the number P(30,25) is so large as to cause the final approximation to be displayed in scientific notation.

Example 4: In how many ways can a dozen soldiers be lined up in a row?

Here we need to compute the number of permutations of 12 objects taken 12 at a time. Key 12 ENTER↑ XEQ "PM". P(12,12) = 479,001,600.

FORMULAS USED IN PM

PM calculates the number of permutations of n objects taken k at a time by the following formula:

$$P(n,k) = n!/(n-k)! = n(n-1)(n-2)(n-3)\cdots(n-(k+1))$$

Routine Listing For: PM

```
80♦LBL C
81♦LBL "PM"
82 CHS
83 X<>Y
84 SIGN
85 X<> L
86 ST+ Y
87♦LBL 06
88 X=Y?
89 GTO 07
90 ST* L
91 DSE X
92 GTO 06
93♦LBL 07
94 RDN
95 X<> L
96 RTN
```

LINE BY LINE ANALYSIS OF PM

Lines 80-86 initialize the program by storing 1 in LAST X and storing n in X (line 84), and storing the ending test value n-k in Y (line 86).

Lines 87-92 are the main loop in the program which simply multiply LAST X by the counter value in the X register. This counter starts with n and decreases by one each time through the loop until the value n-k is reached. Line 88 tests when to exit the loop.

Lines 93-96 recall the product from LAST X which is the final answer. The purpose of line 94 is to drop Z and T which the PM routine preserves from when it is called. Otherwise line 94 would not be needed. P(n,k) is left in the X-register with the original Z and T contents returned in Y and Z respectively.

CONTRIBUTORS HISTORY FOR PM

The PM routine and documentation were written by John Kennedy (918) with help from Keith Jarrett (4360).

FINAL REMARKS FOR PM

Future versions of PM may be able to extend the range of input values to include zero as a valid argument. This feature was not included in the PPC ROM due to limited space.

FURTHER ASSISTANCE ON PM

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

NOTES

TECHNICAL DETAILS

XROM: 20, 19	**PM**	SIZE: 000 minimum

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:

R00: not used

R06: **PM** does not use
R07: any data registers
R08: all computations
R09: are carried out
R10: in the stack
R11:
R12:

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode:
not used
FIX 0 recommended

Angular Mode:
not used

Unused Subroutine Levels:
5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine:
C 06, 07

Execution Time: data dependent with range less than one second to over five seconds

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: **BD**

Bytes In RAM: 32

Registers To Copy: 53

Other Comments:

OUT OF RANGE message indicates too large inputs. SCI display mode indicates overflow but may still give valid approximation

PO - PAPER OUT

PO is simply five paper advances and was included as a "convenience routine."

Example 1: A PPC member wants to document a group of data registers by taping the printer output in his notebook. He uses BV to list the registers.

```
XROM PO
 1.01
XROM BV
XROM PO
XROM PO
```

COMPLETE INSTRUCTIONS FOR PO

PO may be executed any time. No inputs are required and while PO is a peripheral routine normally used with the 82143A Peripheral Printer, it will not stop if the printer is not connected.

MORE EXAMPLES OF PO

EXAMPLE 2: A PPC member wants to insure that the PPC ROM is plugged in before he runs a long program. To insure this he places XEQ PO as his first program instruction following his label. If the PPC ROM is not plugged in an immediate NONEXISTENT appears.

EXAMPLE 3: A rapid alarm is desired for a games program. A standard TONE 9 loop of LBL 01, TONE 9, GTO 01 is too fast--about four per second. If an XEQ PO is added to the loop a two TONES per second rate is obtained.

BACKGROUND FOR PO

One of the goals we had in "programming" the PPC 8K ROM was to use every byte possible. The limitations of the SDS (SDS I) system made arbitrary XROM calls across the 4K "boundary" difficult. The result was always a few odd bytes left over that should obviously be used. A collection of short routines was kept in reserve to be added at the last minute if space permitted. PO, AM, MA, T2, T3, and a few other tones were among these routines. The tone routines didn't make the ROM.

FURTHER DISCUSSION OF PO

PO may be used with or without the printer. It may be used to provide a non-stack upsetting delay, a PPC ROM test, a routine to confuse the non-PPC member, and a means to move the paper out of the printer. A loop containing XROM PO five times takes about one second. PO was added to the ROM during the foggy early morning hours of SDS "programming". It is a practical "routine" and certainly looks better in a program than long strings of advances. If the "delay feature" of PO is used with a printer flag 21 may be cleared before PO is executed and set again after PO is executed.

Routine Listing For:	PO
27♦LBL "PO"	31 ADV
28 ADV	32 ADV
29 ADV	33 RTN
30 ADV	

TECHNICAL DETAILS

XROM: 20,51 — **PO** — SIZE: 000

Stack Usage: NONE	Flag Usage: NONE
0 T:	04:
1 Z:	05:
2 Y:	06:
3 X:	07:
4 L:	08:
Alpha Register Usage:	09:
5 M:	10:
6 N: NONE	
7 O:	
8 P:	25:
Other Status Registers:	**Display Mode:** N/A
9 Q:	
10 ⊦:	
11 a: NONE USED BY	**Angular Mode:** N/A
12 b: ROUTINE	
13 c:	
14 d:	**Unused Subroutine Levels:**
15 e:	5
ΣREG: NOT USED	**Global Labels Called:** Direct / Secondary
Data Registers:	NONE / NONE
R00:	
R06: NONE USED BY	
R07: ROUTINE	
R08:	
R09:	
R10:	
R11:	**Local Labels In This Routine:**
R12:	NONE

Execution Time: 2.4 seconds.*

Peripherals Required: 82143A PRINTER

Interruptible? YES	**Other Comments:**
Execute Anytime? YES	*Approximately 120 Ms. if printer is not connected.
Program File: NS	
Bytes In RAM: 14	
Registers To Copy: 16	

CONTRIBUTORS HISTORY FOR PO

This brilliant Routine obviously required a major time contribution of all 4,000 PPC members to program.

APPENDIX J CONTINUED FROM PAGE 349.

```
172 CLX
173 X<> \
174 "├**"
175 STO [
176 "├**"
177 X<> \
178 CLA
179 X<> [
180 RDN

181•LBL "Ab"
182 ASTO b

183•LBL 14
184 RCL c
185 STO [
186 "├******"
187 X<> [
188 X<> d
189 CF 00
190 CF 01
191 CF 02
192 CF 03
193 X<> d
194 RTN

195•LBL "E?"
196 RCL c
197 XROM "2D"
198 16
199 MOD
200 LASTX
201 X↑2
202 *
203 RCL [
204 +
205 CLA
206 END
 01•LBL "FI"
02 GTO IND 06

03•LBL e
04•LBL 00
05 E
06 STO 08
07 STO 09
08 CLX
09 STO 01
10 STO 02
11 STO 03
12 STO 04
13 STO 05
14 GTO 10

15•LBL c
16 FC?C 08
17 SF 08
18 GTO 10

19•LBL d
20 FC?C 09
21 SF 09

22•LBL J
23•LBL 10
24 "D"
25 FS? 08
26 "C"
27 FC? 09
28 "├E"
29 FS? 09
30 "├B"
31 ASTO X
32 RTN

33•LBL H
34 STO 08
35 CF 22
36 RTN

37•LBL I
38 STO 09
39 CF 22
40 RTN

41•LBL a
42 12
43 *

44•LBL A
45 FS? 22
46 STO 01
47 FS?C 22
48 RTN

49•LBL 01
50 XEQ 07
51 STO Z
52 RCL 05
53 -
54 R↑
55 RCL 03
56 +
57 /
58 LN
59 RCL 07
60 LN1+X
61 /
62 STO 01
63 RTN

64•LBL b
65 12
66 /

67•LBL B
68 FS? 22
69 STO 02
70 FS?C 22
71 RTN

72•LBL 02
73 RCL 03
74 ABS
75 RCL 05
76 ABS
77 +
78 RCL 04
79 X=0?
80 GTO 09
81 /
82 ABS
83 1/X
84 LASTX
85 RCL 01
86 3
87 Y↑X
88 /
89 +
90 STO 07

91•LBL 06
92 XEQ 08
93 STO 02
94 RCL 03
95 +
96 STO Z
97 X<>Y
98 ST* 02
99 *
100 RCL 03
101 +
102 RCL 05
103 +
104 X<> Z
105 *
106 RCL 07
107 FS? 10
108 VIEW X
109 E
110 +
111 /
112 RCL 01
113 *
114 RCL 02
115 RCL 07
116 /
117 -
118 /
119 ST- 07
120 RCL 07
121 /
122 E2
123 *
124 RND
125 X≠0?
126 GTO 06
127 GTO 11

128•LBL 07
129 E
130 RCL 02
131 %
132 RCL 08
133 RCL 09
134 FS? 08
135 X<>Y
136 RDN
137 /
138 STO 07
139 LN1+X
140 RCL 08
141 RCL 09
142 /
143 *
144 FS? 08
145 X<> 07
146 E↑X-1
147 STO 07

148•LBL 08
149 E
150 RCL 07
151 FS? 09
152 ST+ Y
153 /
154 E
155 RCL 01
156 RCL 07
157 LN1+X
158 *
159 E↑X-1
160 +
161 LASTX
162 RCL 04
163 R↑
164 *
165 RTN

166•LBL 09
167 RCL 05
168 RCL 03
169 /
170 CHS
171 LN
172 RCL 01
173 /
174 E↑X-1
175 STO 07

176•LBL 11
177 CLD
178 RCL 07
179 LN1+X
180 RCL 09
181 *
182 RCL X
183 RCL 08
184 /
185 E↑X-1
186 RCL 08
187 *
188 FS? 08
189 X<>Y
190 E2
191 *
192 STO 02
193 RTN

194•LBL C
195 FS? 22
196 STO 03
197 FS?C 22
198 RTN

199•LBL 03
200 XEQ 07
201 *
202 RCL 05
203 +
204 R↑
205 /
206 CHS
207 STO 03
208 RTN

209•LBL D
210 FS? 22
211 STO 04
212 FS?C 22
213 RTN

214•LBL 04
215 XEQ 07
216 X<> L
217 *
218 CHS
219 RCL 03
220 R↑
221 *
222 RCL 05
223 +
224 X<>Y
225 /
226 STO 04
227 RTN

228•LBL E
229 FS? 22
230 STO 05
231 FS?C 22
232 RTN

233•LBL 05
234 XEQ 07
235 RCL 03
236 +
237 *
238 RCL 03
239 +
240 CHS
241 STO 05

242•LBL 12
243 END
---------------- ROM 10 | ROM 20 ----------------
 01•LBL "SR"
 02 SIGN
 03 SF 10
 04 RDN
 05 RCL b
 06 STO [
 07 RDN
 08 FC?C 10
 09 RTN
 10 "├***"
11 RCL IND L
 12 ISG L
 13 ""
14 X<> IND L
 15 STO \
 16 "├**"
17 X<> IND L
 18 STO ]
 19 "├**"
 20 X<> ]
 21 STO a
 22 X<> \
 23 CLA

 24•LBL "Sb"
 25 STO b

 26•LBL "LR"
 27 SIGN
 28 RDN
 29 "+"
 30 RCL a
 31 STO \
 32 RDN
 33 RCL b
 34 X<> [
 35 STO ]
36 ASTO IND L
 37 ISG L
 38 ""
 39 "├*****"
 40 STO ]
41 ASTO IND L
 42 RDN
 43 CLA
 44 RTN

 45•LBL "SD"
 46 SIGN
 47 RDN
 48 RCL d
 49 STO [
 50 "├**"
 51 X<> [
 52 GTO 14

 53•LBL "SK"
 54 SIGN
 55 CLX
 56 X<> '
 57 XEQ 14
 58 ISG L
 59 ""
 60 .
 61 X<> e

 62•LBL 14
 63 "*"
 64 X<> [
 65 STO \
66 ASTO IND L
 67 RDN
 68 RTN

 69•LBL "RD"
 70 SIGN
71 ARCL IND L
 72 RDN
 73 RCL d
 74 STO \
 75 "├**"
 76 X<> ]
 77 STO \
 78 "├*****"
 79 X<> \
 80 STO d
 81 RDN
 82 CLA
 83 RTN

 84•LBL "RK"
 85 SIGN
86 ARCL IND L
 87 "├*"
 88 ISG L
 89 ""
90 ARCL IND L
 91 "├‡"
 92 X<> \
 93 STO '
 94 X<> [
 95 STO e
 96 RDN
 97 CLA
 98 RTN

 99•LBL "BV"
100 .
101 ENTER↑

102•LBL 00
103 CLX
104 RCL IND Z
105 X=Y?
106 GTO 01
107 X<> Z
108 INT
109 CLA
110 RCL d
111 CF 29
112 FIX 0
113 ARCL Y
114 STO d
115 "├: "
116 R↑
117 ARCL X
118 XROM "VA"
119 FS? 10
120 STOP
121 FS? 09
122 PSE
123 LASTX
124 .
125 ENTER↑

126•LBL 01
127 TONE 8
128 ISG Z
129 GTO 00
130 TONE 6
131 END
 01•LBL "CV"
02 GTO IND 06

 03•LBL A
 04•LBL 01
 05 CF 10

 06•LBL 06
 07 STO 09
 08 X<>Y
 09 STO 08
 10 ΣREG 13
 11 FC? 10
 12 Σ+
 13 FS? 10
 14 Σ-
 15 RDN
 16 RCL 08
 17 ENTER↑
 18 X>0?
 19 LN
 20 ST* Z
 21 RCL 09
 22 X>0?
 23 LN
 24 ST* Z
 25 X<>Y
 26 ΣREG 19
 27 FC? 10
 28 Σ+
 29 FS? 10
 30 Σ-
 31 R↑
 32 FS? 10
 33 CHS
 34 ST+ 12
 35 R↑
 36 FS? 10
 37 CHS
 38 ST+ 11
 39 X<> Z
 40 SIGN
 41 ST+ L
 42 RCL 08
 43 RCL 09
 44 X<> L
 45 RTN
 46 RCL 08
 47 RCL 09

 48•LBL a
 49 SF 10
 50 GTO 06

 51•LBL B
 52•LBL 02
 53 CF 08
 54 CF 09
 55 STO 07
 56 2
 57 X<Y?
 58 SF 09
 59 /
 60 FRC
 61 X=0?
 62 SF 08
 63 8
 64 ST+ 07
65 XEQ IND 07
 66 RCL 17
 67 RCL 13
 68 RCL 15
 69 STO 09
 70 *
 71 RCL 18
 72 /
 73 -
 74 STO 10
 75 RCL 14
 76 RCL 13
 77 X↑2
 78 RCL 18
 79 /
 80 -
 81 STO Z
 82 /
 83 STO 08
 84 RCL 13
 85 *
 86 ST- 09
 87 X<>Y
 88 RCL 16
 89 RCL 15
 90 X↑2
 91 RCL 18
 92 ST/ 09
 93 /
 94 -
 95 *
 96 SQRT
 97 ST/ 10
98 XEQ IND 07
 99 8
100 ST- 07
101 RCL 10
102 RCL 09
103 FS? 08
104 E↑X
105 STO 09
106 RCL 08
107 RTN

108•LBL 10
109 RCL 11
110 X<> 17
111 STO 11

112•LBL 13
113 RCL 21
114 X<> 15
115 STO 21
116 RCL 22
117 X<> 16
118 STO 22

119•LBL 09
120 RTN

121•LBL 11
122 RCL 12
123 X<> 17
124 STO 12

125•LBL 14
126 RCL 19
127 X<> 13
128 STO 19
129 RCL 20
130 X<> 14
131 STO 20
132 RTN

133•LBL 12
134 RCL 23
135 X<> 17
136 STO 23
137 XEQ 14
138 GTO 13

139•LBL C
140•LBL 03
141 FS? 09
142 LN
143 RCL 08
144 *
145 RCL 09
146 FS? 08
147 LN
148 +
149 FS? 08
150 E↑X
151 RTN

152•LBL D
153•LBL 04
154 FS? 08
155 LN
156 RCL 09
157 FS? 08
158 LN
159 -
160 RCL 08
161 /
162 FS? 09
163 E↑X
164 RTN

165•LBL e
166•LBL 00
167 11.024
168 XROM "BC"
169 E
170 RTN

171•LBL E
172•LBL 05
173 .
174 STO 25
175 4
176 STO 07

177•LBL 07
178 RCL 07
179 XEQ B
180 RCL 25
181 RCL 10
182 ABS
183 X<=Y?
184 GTO 15
185 STO 25
186 RCL 07
187 STO 26

188•LBL 15
189 DSE 07
190 GTO 07
191 RCL 26
192 XEQ 02
193 RCL 26
194 END
 01•LBL "IG"
 02•LBL B
 03 STO 17
 04 X<>Y
 05 -
 06 4
 07 /
 08 STO 16
 09 ST- 17
 10 ST- 17
 11 .
 12 STO 15
 13 STO 11
 14 STO 18
 15 SF 09

 16•LBL 01
 17 E
 18 2
 19 STO 14
 20 RCL 11
 21 CHS
 22 Y↑X
 23 ST* 14
 24 E
 25 -

 26•LBL 02
 27 STO 12
 28 X↑2
 29 -
 30 STO 13
 31 2
 32 +
 33 RCL 12
 34 *
 35 RCL 16
 36 *
 37 RCL 17
 38 +
39 XEQ IND 10
 40 RCL 13
 41 *
 42 ST+ 15
 43 E
 44 RCL 12
 45 RCL 14
 46 +
 47 X<Y?
 48 GTO 02
 49 RCL 11
 50 STO 13
 51 18
 52 STO 12
 53 E
 54 ST+ 11
 55 RCL 15
 56 RCL 16
 57 1.5
 58 *
 59 *
 60 RCL 14
 61 *

 62•LBL 03
 63 R↑
 64 4
 65 *
 66 ENTER↑
 67 DSE Y
 68 X<> Z
 69 ENTER↑
70 X<> IND 12
 71 ST- Y
 72 RND
 73 X<> Z
 74 /
75 RCL IND 12
 76 +
 77 ISG 12
 78 STOP
 79 DSE 13
 80 GTO 03
81 STO IND 12
 82 FS? 10
 83 VIEW X
 84 FS?C 09
 85 GTO 01
 86 RND
 87 X≠Y?
 88 GTO 01
 89 LASTX
 90 RTN

 91•LBL C
 92•LBL "SV"
 93 STO 07
 94 E
 95 %
 96 RCL Z
 97 X=0?
 98 X<>Y
 99 STO 09
100 CLST

101•LBL 04
102 RCL Z
103 STO 08
104 RCL 07
105 FS? 10
106 VIEW X
107 XEQ IND 06
108 ST* 09
109 ST- 08
110 RCL 09
111 RCL 08
112 X≠0?
113 /
114 STO 09
115 X<> 07
116 ST+ 07
117 RND
118 RCL 07
119 RND
120 X≠Y?
121 GTO 04
122 RCL 07
123 RTN

124•LBL D
125•LBL "FD"
126 FS? 09
127 GTO 08
128 17
129 XROM "SD"
130 SCI 1
131 2 E-3
132 STO 14

133•LBL 05
134 RCL 12
135 .7
136 *
137 RND
138 STO 12
139 XEQ 08
140 ENTER↑
141 X<> 16
142 -
143 ENTER↑
144 FS? 10
145 VIEW X
146 X<> 15
147 ISG 14
148 GTO 05
149 LASTX
150 RDN
151 X=0?
152 GTO 07
153 /
154 E
155 X<>Y
156 X<0?
157 GTO 06
158 X<Y?
159 GTO 05

160•LBL 06
161 X<> L

162•LBL 07
163 R↑
164 .7
165 ST/ 12
166 CLX
167 10
168 XROM "RD"
169 RTN

170•LBL 08
171 .
172 STO 13
173 XEQ IND 10
174 11
175 XEQ 09
176 -18
177 XEQ 09
178 9
179 XEQ 09
180 ST+ X
181 RCL 13
182 -
183 RCL 12
184 3
185 *
186 ST- IND 11
187 ST+ X
188 /
189 ENTER↑
190 RTN

191•LBL 09
192 *
193 ST+ 13
194 RCL 12
195 ST+ IND 11
196 GTO IND 10
197 END
 01•LBL "FR"
02 GTO IND 06

 03•LBL B
 04•LBL 02
 05 CHS

 06•LBL A
 07•LBL 01
 08 ST* T
 09 X<> Z
 10 *
 11 ST+ Z
 12 X<> L
 13 *
 14 GTO 05

 15•LBL D
 16•LBL 04
 17 X<>Y

 18•LBL C
 19•LBL 03
 20 ST* Z
 21 X<> T
 22 *
 23 X<>Y

 24•LBL E
 25•LBL 05
 26 RCL Y
 27 RCL Y
 28 XEQ 06
 29 ST/ Z
 30 /
 31 FC? 10
 32 RTN
 33 FIX 0
 34 " "
 35 ARCL Y
 36 "├/"
 37 ARCL X
 38 XROM "VA"
 39 RTN

 40•LBL c
 41•LBL 06
 42 MOD
 43 LASTX
 44 X<>Y
 45 X≠0?
 46 GTO c
 47 +
 48 RTN

 49•LBL d
 50•LBL "DF"
 51 STO 08
 52 INT
 53 .
 54 STO 09
 55 E
 56 STO 10
 57 RCL 08
 58 R↑
 59 X=Y?
 60 GTO 08
 61 ST- Y

 62•LBL 07
 63 RDN
 64 1/X
 65 ENTER↑
 66 INT
 67 -
 68 RCL 09
 69 RCL 10
 70 STO 09
 71 LASTX
 72 *
 73 +
 74 STO 10
 75 RCL 08
 76 *
```

APPENDIX J CONTINUED ON PAGE 373.

PR - PACK REGISTER

This routine is called pack register and can be used to store data in packed form in a data register. The packing scheme is to simply encode data assuming a base b representation that is usually other than base 10. This routine first appeared on the HP-67/97 in the booklet BETTER PROGRAMMING ON THE HP-67/97. See also the routine UR. PR calls the UR routine. Using base b data packing techniques it is possible to store several numbers in one register. The PR routine is used to store numbers in packed form in a data register.

Example 1: Use the base b=52 and the register R15 to hold the five numbers 46, 18, 44, 38, and 29 in packed form. Use PR to store these numbers in R15.

As does UR, the PR routine assumes that the base b is stored in R10 and that R11 contains the number of the data register that will be packed. For this example store the following data.

R10: 52 = base b
R11: 15 = pointer to register R15

To understand the packing routine consider the number which is formed by using the above 5 numbers as coefficients on powers of 52.

$$29*52^4+38*52^3+44*52^2+18*52^1+46*52^0 = 217,499,926.$$

The five coefficients are assumed to be numbered from 1-5 starting with the zero power of 52. The powers of 52 range from 0-4 but the PR (and UR) routine assumes the corresponding range to be 1-5. In this example we can see the correspondence between the position numbers and the stored data.

The number 29 corresponds to position 5.
The number 38 corresponds to position 4.
The number 44 corresponds to position 3.
The number 18 corresponds to position 2.
The number 46 corresponds to position 1.

Provided that R10 and R11 have been initialized with the above data it is a simple matter to use PR to store the five numbers in R15. Store 0 in R15.

Key in each number followed by its position number and XEQ "PR".

To store 46 in position 1 key 46 ENTER↑ 1 and XEQ "PR".

To store 18 in position 2 key 18 ENTER↑ 2 and XEQ "PR".

To store 44 in position 3 key 44 ENTER↑ 3 and XEQ "PR".

To store 38 in position 4 key 38 ENTER↑ 4 and XEQ "PR".

To store 29 in position 5 key 29 ENTER↑ 5 and XEQ "PR".

Now recall the contents from register 15. You should see the number 217,499,926 in R15.

COMPLETE INSTRUCTIONS FOR PR

1) PR assumes that R10 holds the base b and that R11 is pointing to the register that will store the data to be packed.

R10: base b
R11: register pointer

2) To store the number n in position k, key n ENTER↑ k and XEQ "PR". The data is stored in the register pointed to by R11. The number n must be in the range 0-(b-1). PR calls UR and does not return any useful values in the stack. The stack input/output for PR is as follows:

Input:		Output:	
	T: T		T: n
	Z: Z		Z: n
	Y: number n		Y: *
	X: position k		X: nb^{k-1}
	L: L		L: b^{k-1}

The following table indicates the range of possible bases and position numbers.

Data Range	Base b	Position Numbers
0-1	2	1-30
0-2	3	1-19
0-3	4	1-15
0-4	5	1-13
0-6	7	1-11
0-9	10	1-10
0-13	14	1-8
0-20	21	1-7
0-36	37	1-6
0-99	100	1-5
0-214	215	1-4
0-1413	1414	1-3
0-99999	100000	1-2

The most efficient use may be made of data registers by storing the largest data values in the lowest numbered positions and storing the smallest data values in the highest numbered positions. If your priority is the range of data, start with the column on the left. If your priority is the number of artificial memories available, start with the column on the right. In many cases it will be possible to extend the values in this table.

MORE EXAMPLES OF PR

Example 2: From the above table it can be seen that when the base b=21 we may store as many as 7 numbers in one register provided the numbers are in the range 0-20. Use PR to pack the numbers 13, 19, 14, 15, 8, 18, and 16 all in register 12.

First store the base 21 in R10 and store the number 12 in R11. We will use PR to pack the above numbers one by one starting with position 1. The position numbers range from 1 to 7. Store 0 in R12

To store 13 in position 1 key 13 ENTER↑ 1 XEQ "PR".
To store 19 in position 2 key 19 ENTER↑ 2 XEQ "PR".
To store 14 in position 3 key 14 ENTER↑ 3 XEQ "PR".
To store 15 in position 4 key 15 ENTER↑ 4 XEQ "PR".
To store 8 in position 5 key 8 ENTER↑ 5 XEQ "PR".
To store 18 in position 6 key 18 ENTER↑ 6 XEQ "PR".
To store 16 in position 7 key 16 ENTER↑ 7 XEQ "PR".

Now recall R12 and see the number 1,447,473,103.

The base 21 representation of this number shows the seven numbers as coefficients on powers of 21.

1,447,473,103 =

$16*21^6 + 18*21^5 + 8*21^4 + 15*21^3 + 14*21^2 + 19*21 + 13$

Routine Listing For: PR

```
230♦LBL "PR"
231 XROM "UR"
232 X<>Y
233 ST* Z
234 *
235 ST- IND 11
236 X<>Y
237 ST+ IND 11
238 RTN
239 END
```

LINE BY LINE ANALYSIS OF PR

Line 231 calls the UR routine so the number in the present position can be recalled.

Lines 232 and 233 preserve a power of the base that is used to multiply both the number recalled and the number to be stored.

Line 235 serves to clear the position to be occupied by the new number which is stored in the desired position at line 237.

REFERENCES FOR PR

John Kennedy, "Data Packing," BETTER PROGRAMMING ON THE HP-67/97," by Bill Kolb (265), Richard Nelson (1), and John Kennedy (918).

CONTRIBUTORS HISTORY FOR PR

PR and the corresponding documentation were written by John Kennedy (918).

FURTHER ASSISTANCE ON PR

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

TECHNICAL DETAILS

XROM: 20, 45 — **PR** — **SIZE:** depends on registers used

Stack Usage:
0 T: used
1 Z: used
2 Y: used
3 X: used
4 L: used

Alpha Register Usage:
5 M: not used
6 N: not used
7 O: not used
8 P: not used

Other Status Registers:
9 Q: not used
10 ⊦: not used
11 a: not used
12 b: not used
13 c: not used
14 d: not used
15 e: not used

ΣREG: not used

Data Registers:
R00: not used
R06: not used
R07: not used
R08: not used
R09: not used
R10: base b
R11: pointer to data reg.
R12: not used

Flag Usage:
04: not used
05: not used
06: not used
07: not used
08: not used
09: not used
10: not used
25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 4

Global Labels Called:
Direct: UR
Secondary: none

Local Labels In This Routine: none

Execution Time: 1.5 seconds

Peripherals Required: none

Interruptible? yes
Execute Anytime? no
Program File: M2
Bytes In RAM: 21
Registers To Copy: 61

Other Comments: Any registers used to store data should be cleared to 0 before first being used.

PS - PAGE SWITCH

PS provides a memory paging capability for use with a port extender, enabling the user to switch memory modules on and off without disrupting the calculator. (See the **IP** writeup for an explanation of why and how this is accomplished.) **PS** stores the contents of status register c in register 256, the bottom register of the memory module, then prompts the user to switch modules. When the new module is activated its bottom register (location 256) is recalled and placed in c. Clearly setup is required before this switching technique can work. All of the modules but one must be initialized by placing the c register contents in location 256. The initialization procedure uses **IP** and is described in the **IP** writeup.

Example 1: Suppose you have set up three modules, numbered 1, 13, and 45, using the procedure given in Example 1 of the **IP** writeup. Module 45 is on line after this setup procedure is complete. To switch from module 45 to module 13 and pick up execution at LBL "XYZ", use the following sequence:

```
CF 10
45
ENTER↑
13
"XYZ"
XEQ PS
```

This sequence can be keyed in or it can be executed as part of a program. **PS** will respond with a prompt:

"45 OFF, 13 ON'
Switch module 45 off line and 13 on line, then press R/S.
Execution is automatically resumed at LBL "XYZ" in Module 13.

COMPLETE INSTRUCTIONS FOR PS

To switch among N modules, use **IP** on the first N-1 of them as described in the **IP** writeup.

To switch from module n to module m and resume execution at LBL "ABCDEF" in module m, make sure flag 10 is clear, then use the sequence:

```
n
ENTER↑
m
"ABCDEF"
XEQ PS
```

The entries n and m are integers between 0 and 127. Numbers up to 999 can be used if the TONE option is not chosen. The label name ("ABCDEF" here) must not exceed six characters.

The above sequence can be keyed in manually or included in a program. In either case, **PS** will respond with the message "n off, m on". Turn off module n and turn on module m, then press R/S to continue. **PS** will set up the pointers for module m and jump to "ABCDEF".

If you have a tone-controlled device capable of toggling modules on and off in response to selected TONEs (normal or synthetic), you can make use of the automatic TONE control option of **PS**. Simply set flag 10 and XEQ **PS** with the same inputs as before. Instead of prompting you to switch modules, the 41C will produce TONE n, then TONE m. The tone decoder should turn module n off and module m on in response to these tones. The module numbers n and m can be chosen to meet the tone decoder's requirements for discrimination between TONEs.

The flag 24 option of **PS** is provided to allow future use of a software-controlled page selector. The RTN on line 102 can be synthetically set up to return control to a user program below the .END. and above the key assignments or even in an external EPROM box. This user program can then select another module (by methods yet unknown) and branch to the designated label in the new module.

Since the .END. must be outside the mainframe memory to use **PS**, carrying data from one page to another would appear to be out of the question. However, PPC ROM routine **SX** provides access to the free registers between the top of the key assignments and the .END. of program memory (see the memory map in the **LF** writeup). The allowable addresses for this purpose are 193 + INT (**A?** + .5) through 255.

To store Y in absolute address X use Y ↑ X XEQ **SX**. To recall and clear absolute address X use 0 ↑ X XEQ **SX** followed by X<>Z. If you need to carry a large block of data when page switching use **OM** and X<> IND for greater speed.

Routine Listing For: PS

```
76♦LBL "PS"          109 RDN
77 E3                110 CLX
78 /                 111 X<> IND T
79 +                 112 STO c
80 ASTO Y            113 XEQ IND Z
81 XEQ 14            114 XROM "GE"
82 FRC
83 E3                115♦LBL 14
84 ST* Y             116 XROM "E?"
85 RDN               117 257
86 RCL d             118 -
87 FIX 0             119 X<0?
88 CF 29             120 GTO 14
89 " "               121 R↑
90 ARCL L            122 SIGN
91 "⊦ OFF;"          123 R↑
92 ARCL Y            124 R↑
93 "⊦ ON"            125 ΣREG T
94 STO d             126 XROM "OM"
95 X<> L             127 STO [
96 INT               128 240
97 FC? 10            129 ASTO IND X
98 PROMPT            130 R↑
99 FC? 10            131 R↑
100 GTO 13           132 RTN
101 FS? 24
102 RTN              133♦LBL 14
103 XROM "TN"        134 ENTER↑
104 XROM "TN"        135 XROM "S?"
105 R↑               136 +
106 R↑               137 "OVERSIZE"
                     138 XROM "VA"
107♦LBL 13           139 XROM "GE"
108 240
```

TECHNICAL DETAILS

XROM: 10,46	PS	SIZE: 000

Stack Usage:

0 T: 240
1 Z: DESTINATION LABEL
2 Y: X
3 X: NEW c REGISTER
4 L: Y.X

Alpha Register Usage:

5 M:
6 N: REPLACED BY
7 O: SWITCHING PROMPT
8 P:

Other Status Registers:

9 Q: NOT USED
10 ⊢: NOT USED
11 a: NOT USED
12 b: NOT USED
13 c: TAKEN FROM NEW PAGE
14 d: USED BUT RESTORED
15 e:

ΣREG: SET TO 256 ABSOLUTE

Data Registers:

R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Absolute location 256 (below the .END.) is used in the old page and recalled and cleared in the new page.

Flag Usage: SEVERAL USED BUT ALL RESTORED

04:
05:
06:
07:
08:
09:
10: TONE CONTROL IF SET (AND F24 CLEAR)
24: SOFTWARE CONTROL IF SET (AND F10 SET)
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 0

Global Labels Called:

Direct	Secondary
TN	DC
E?	XE
OM	2D
S?	C?
VA	
GE	

Local Labels In This Routine:

13
14 TWICE

Execution Time: 5 seconds.

Peripherals Required: TWO MEMORY MODULES

Interruptible? YES
Execute Anytime? NO
Program File: BL
Bytes In RAM: 126
Registers To Copy: 46

Other Comments:

LINE BY LINE ANALYSIS OF PS

Lines 76-79 combine the two module numbers into a single constant in order to conserve stack space. The destination label is also stored in the stack. Line 81 calls a routine (lines 115-139) which stores the old c register as an alpha constant in location 256. This routine is described in the IP writeup. Lines 82-84 separate the two module numbers, while lines 85-96 construct the message to switch from one module to the other and place the two module numbers in X and Y. Lines 97-106 prompt if flag 10 is clear (normal mode), produce two tones for switching if flag 10 is set and flag 24 is clear (tone control mode) or execute a RTN to a user program segment in ROM or in the mainframe RAM (below the .END. and above the key assignment area) if flags 10 and 24 are set (software control mode). Lines 107-112 extract the prestored c register information from the new module and place it in the c register. Line 113 transfers execution to the specified destination label. Line 114 is provided in case that destination program ends in RTN or END, rather than STOP or XEQ PS.

REFERENCES FOR PS

See *PPC CALCULATOR JOURNAL*, V8N1P25.

CONTRIBUTORS HISTORY FOR PS

Roger Hill (4940) and Keith Jarett (4360) wrote the final ROM version, but Richard Nelson (1), Lee Vogel (4196) and others made valuable suggestions.

FURTHER ASSISTANCE ON PS

Call Roger Hill (4940) at (618) 656-8825 or during some holiday periods at (213) 794-7376.

FURTHER ASSISTANCE ON PS

Call Roger Hill (4940) at (618) 656-8825.
Call Richard Nelson (1) at (714) 754-6226.

NOTES

QR - QUOTENT REMAINDER

QR is a complete MOD function, providing not only y mod x, but also the quotient, (y-y mod x)/x. This nifty little routine is essentially a poor man's base conversion. It is used by many of the PPC ROM routines, notably in synthetic routines that need to slice bytes into nybbles (using x=16). It's a real workhorse routine that more than paid for itself in byte savings in other routines.

Example 1: Find the hexadecimal (base 16) components of 214. Key in 214 ENTER↑ 16 XEQ QR . The result is 6 in X and 13 (=D_{16}) in Y. Thus 214_{10} = $D6_{16}$.

COMPLETE INSTRUCTIONS FOR QR

Just XEQ QR . Y is replaced by (Y-Y mod X)/X (the quotient) and X is replaced by Y mod X (the remaider). Z and T are preserved, and the old X is placed in L. The alpha register is preserved if it contained no more than 14 characters (status register 0 is cleared). Zero in X causes DATA ERROR.

APPLICATION PROGRAM 1 FOR QR

QR can be used repeatedly to decompose any number into base X digits. This program "YBX" (Y base X) performs such a decomposition, producing digits one by one from least significant to most significant. Y and X must be integers.

```
LBL "YBX"
SIGN
LBL 00
X<>L
XROM QR
STOP
CLX
X≠Y?
GTO 00
XROM T1
```

For example 1103 ENTER↑ 8 XEQ "YBX" produces 7 (R/S) 1 (R/S) 1 (R/S) 2 (R/S) 0. Thus 1103_{10} = 2117_8. You can check this answer using the built in functions OCT and DEC.

Routine Listing For: QR	
82♦LBL "QR"	88 LASTX
83 X<>Y	89 ST/]
84 STO]	90 CLX
85 X<>Y	91 X<>]
86 MOD	92 X<>Y
87 ST-]	93 RTN

LINE BY LINE ANALYSIS OF QR

Lines 82-86 store Y in status register 0 and takes Y mod X. This quantity is subtracted from Y (line 87) and divided by X (line 89). Then 0 is cleared and the quotient is removed (lines 90-91). The quotient and remainder are interchanged, completing the routine.

CONTRIBUTORS HISTORY FOR QR

The first QR routine (see *PPC CALCULATOR JOURNAL*, V6N8P26c) was written by John Kennedy (918). It was a non-synthetic version that lost the T register and did not save X in LastX. The ROM version was written by Roger Hill (4949). It solved the problems of the earlier version by using status register 0.

FURTHER ASSISTANCE ON QR

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,54 — QR — SIZE: 000

Stack Usage:	Flag Usage: NONE USED
0 T: T	04:
1 Z: Z	05:
2 Y: (Y-Y MOD X)/X	06:
3 X: Y MOD X	07:
4 L: X	08:
Alpha Register Usage:	09:
5 M: UNCHANGED	10:
6 N: UNCHANGED	
7 O: CLEARED	
8 P: UNCHANGED	25:
Other Status Registers:	Display Mode: UNCHANGED
9 Q:	
10 ⊦: NONE USED	
11 a:	Angular Mode: UNCHANGED
12 b:	
13 c:	
14 d:	Unused Subroutine Levels: 6
15 e:	
ΣREG: UNCHANGED	Global Labels Called: Direct NONE; Secondary NONE
Data Registers: NONE USED	Local Labels In This Routine: NONE

Execution Time: .5 seconds.

Peripherals Required: NONE

Interruptible? YES	Other Comments:
Execute Anytime? NO	
Program File: IF	
Bytes In RAM: 21	
Registers To Copy: 60	

APPENDIX J - PPC CUSTOM ROM LISTING

APPENDIX J CONTINUED FROM PAGE 367.

```
 77 FIX 0
 78 RND
 79 STO Z
 80 RCL 10
 81 /
 82 RCL 08
 83 -
 84 FIX IND 07
 85 RND
 86 X≠0?
 87 GTO 07
 88 RCL Z

 89♦LBL 08
 90 RCL 10
 91 SIGN
 92 ST* 10
 93 *
 94 RCL 10
 95 FC? 10
 96 RTN
 97 GTO 05

 98♦LBL e
 99♦LBL "NP"
100 RCL Y
101 SQRT
102 LASTX
103 X<> Z

104♦LBL 09
105 X>Y?
106 R↑
107 R↑
108 X<>Y
109 MOD
110 X=0?
111 GTO 10
112 X<> L
113 2
114 X=Y?
115 SIGN
116 +
117 GTO 09

118♦LBL 10
119 R↑
120 LASTX
121 X>Y?
122 ENTER↑
123 RTN
124 ST/ Y
125 GTO e

126♦LBL a
127♦LBL "GN"
128 XEQ b
129 LN
130 ST+ X
131 CHS
132 SQRT
133 X<>Y
134 XEQ b
135 360
136 *
137 R↑
138 RCL 07
139 *
140 P-R
141 RCL 06
142 ST+ Z
143 +
144 RTN

145♦LBL b
146♦LBL "RN"
147 RCL IND X
148 9821
149 *
150 .211327
151 +
152 FRC
153 STO IND Y
154 END
 01♦LBL "BD"
 02♦LBL A
 03 CLST

 04♦LBL 01
 05 "├ "
 06 X<> ]
 07 X=0?
 08 GTO 01
 09 X<> [
 10 R↑
 11 X<> \
 12 "├♦Δ"
 13 X<> \
 14 RDN
 15 X<> [
 16 E
 17 *
 18 39
 19 -
 20 X>0?
 21 DSE X
 22 9
 23 +
 24 X<0?
 25 GTO 02
 26 X<>Y
 27 RCL 06
 28 *
 29 +
 30 .
 31 GTO 01

 32♦LBL 02
 33 RDN
 34 CLA
 35 RTN

 36♦LBL B
 37♦LBL "TB"
 38 ""
 39 RCL [
 40 X<>Y

 41♦LBL 03
 42 ENTER↑
 43 INT
 44 RCL 06
 45 MOD
 46 9
 47 -
 48 X>0?
 49 ISG X

 50♦LBL 04
 51 39
 52 +
 53 10↑X
 54 STO ]
 55 "├ "
 56 CLX
 57 X<> ]
 58 X<> \
 59 STO [
 60 RDN
 61 RCL 06
 62 /
 63 INT
 64 X≠0?
 65 GTO 03

 66♦LBL 05
 67 "├ "
 68 CLX
 69 RCL [
 70 X≠Y?
 71 GTO 05
 72 CLX
 73 X<> ]
 74 X<> \
 75 STO [
 76 CLST
 77 FS? 10
 78 XROM "VA"
 79 RTN

 80♦LBL C
 81♦LBL "PM"
 82 CHS
 83 X<>Y
 84 SIGN
 85 X<> L
 86 ST+ Y

 87♦LBL 06
 88 X=Y?
 89 GTO 07
 90 ST* L
 91 DSE X
 92 GTO 06

 93♦LBL 07
 94 RDN
 95 X<> L
 96 RTN

 97♦LBL D
 98♦LBL "CM"
 99 RCL Y
100 RCL Y
101 X≠Y?
102 -
103 X>Y?
104 X<>Y
105 ST+ T
106 SIGN
107 X<>Y

108♦LBL 08
109 X<> T
110 LASTX
111 ST- Y
112 /
113 ST* Y
114 DSE L
115 GTO 08
116 RDN
117 RTN

118♦LBL E
119♦LBL "CJ"
120 INT
121 X<>Y
122 INT
123 2.85
124 -
125 12
126 /
127 R↑
128 INT
129 +
130 X<0?
131 SQRT
132 ENTER↑
133 INT
134 ST- Z
135 X<>Y
136 367
137 *
138 INT
139 ST+ Z
140 SIGN
141 FS? 10
142 ISG X
143 %
144 INT
145 .75
146 ST* Z
147 *
148 RDN
149 -
150 INT
151 R↑
152 -
153 INT
154 1721115
155 +
156 RTN

157♦LBL e
158♦LBL "JC"
159 INT
160 1721119.2
161 -
162 ENTER↑
163 FS? 10
164 -2
165 FS? 10
166 GTO 09
167 36524.25
168 /
169 INT
170 ST+ Y
171 4
172 /
173 INT

174♦LBL 09
175 -
176 X<0?
177 SQRT
178 STO Y
179 365.25
180 ST/ Y
181 X<>Y
182 INT
183 ST* Y
184 RDN
185 INT
186 -
187 .3
188 -
189 STO Y
190 30.6
191 ST/ Y
192 X<>Y
193 INT
194 *
195 ST- Y
196 ISG Y
197 X<> L
198 -3
199 X↑2
200 X<Y?
201 ISG T
202 X<> L
203 -
204 X<>Y
205 INT
206 END
 01♦LBL "CA"
 02 GTO IND 06

 03♦LBL A
 04♦LBL 01
 05 XEQ 16

 06♦LBL 00
 07 XEQ 13
 08 ST+ Z
 09 X<> T
 10 +
 11 X<>Y
 12 GTO 10

 13♦LBL B
 14♦LBL 02
 15 XEQ 16
 16 X<>Y
 17 CHS
 18 X<>Y
 19 CHS
 20 GTO 00

 21♦LBL C
 22♦LBL 03
 23 XEQ 16

 24♦LBL 17
 25 XEQ 13
 26 STO [
 27 X<> T
 28 ST* [
 29 X<>Y
 30 *
 31 X<>Y
 32 LASTX
 33 X<>Y
 34 ST* T
 35 *
 36 RCL [
 37 +
 38 R↑
 39 RCL Z
 40 -
 41 GTO 10

 42♦LBL D
 43♦LBL 04
 44 XEQ 16
 45 STO Z
 46 X↑2
 47 RCL Y
 48 X↑2
 49 +
 50 ST/ Z
 51 /
 52 CHS
 53 X<>Y
 54 GTO 17

 55♦LBL E
 56♦LBL 05
 57 XEQ 16
 58 R-P
 59 LN
 60 GTO 10

 61♦LBL 06
 62 9.009
 63 STO 09
 64 RTN

 65♦LBL 07
 66 2
 67 RCL Z
 68 ST+ X
 69 E↑X-1
 70 +
 71 LASTX
 72 R↑
 73 CHS
 74 E↑X
 75 ST* Z
 76 *
 77 2
 78 ST/ Z
 79 /
 80 RTN

 81♦LBL H
 82♦LBL 08
 83 XEQ 16
 84 XEQ 07
 85 R↑
 86 COS
 87 *
 88 X<>Y
 89 R↑
 90 SIN
 91 *
 92 GTO 10

 93♦LBL I
 94♦LBL 09
 95 XEQ 16
 96 XEQ 07
 97 R↑
 98 SIN
 99 *
100 CHS
101 X<>Y
102 R↑
103 COS
104 *
105 GTO 10

106♦LBL a
107♦LBL 11
108 XEQ 13
109 XEQ 13
110 R↑
111 R↑
112 XEQ 10
113 R↑
114 R↑
115 GTO 10

116♦LBL b
117♦LBL 12
118 XEQ 11
119 XEQ 05
120 XEQ 03

121♦LBL e
122♦LBL 15
123 XEQ 16
124 E↑X
125 P-R
126 GTO 10

127♦LBL 16
128 SF 10

129♦LBL c
130♦LBL 13
131 RCL IND 09
132 FS? 10
133 STO 08
134 DSE 09
135 RCL IND 09
136 FS?C 10
137 STO 07
138 X<>Y
139 DSE 09
140 RTN
141 ISG 09
142 ""
143 ISG 09
144 ""
145 RTN

146♦LBL d
147♦LBL 14
148 RCL 07
149 RCL 08

150♦LBL J
151♦LBL 10
152 X<>Y
153 ISG 09
154 ""
155 STO IND 09
156 X<>Y
157 ISG 09
158 ""
159 STO IND 09
160 END
 01♦LBL "LG"
 02 "01(zΩf)£=x"
 03 "├Ω≠ N"
 04 X<> ]
 05 ACSPEC
 06 X<> \
 07 ACSPEC
 08 X<> [
 09 ACSPEC
 10 X<> ]
 11 RTN

 12♦LBL "HA"
 13 RCL 01
 14 X>Y?
 15 X<>Y
 16 RCL 00
 17 -
 18 RCL 01
 19 RCL 00
 20 -
 21 /
 22 X<0?
 23 .
 24 RCL 02
 25 INT
 26 ST* Y
 27 X<>Y
 28 FIX 4
 29 RND
 30 X<>Y
 31 RCL 02
 32 FRC
 33 E3
 34 *
 35 X<=Y?
 36 X<>Y
 37 RDN
 38 X>Y?
 39 X<>Y
 40 STO Z
 41 -
 42 STO 04
 43 RDN
 44 E
 45 -
 46 SKPCOL
 47 E

 48♦LBL "HS"
 49 RCL 04
 50 *
 51 LASTX
 52 X>Y?
 53 X<>Y
 54 INT
 55 7 E-5
 56 +
 57 RCL 03
 58 GTO 00

 59♦LBL 01
 60 ACCHR

 61♦LBL 00
 62 DSE Y
 63 GTO 01
 64 RDN
 65 INT
 66 8
 67 +
 68 RCL 05
 69 GTO 00

 70♦LBL 02
 71 ACCOL

 72♦LBL 00
 73 DSE Y
 74 GTO 02
 75 RTN

 76♦LBL "CP"
 77 RND
 78 RCL 06
 79 RCL Y
 80 ABS
 81 X≠0?
 82 LOG
 83 INT
 84 9
 85 FS? 40
 86 X<Y?
 87 GTO 06
 88 RDN
 89 X<0?
 90 CLX
 91 FC? 29
 92 GTO 04
 93 .75
 94 /
 95 INT

 96♦LBL 04
 97 -
 98 SKPCHR
 99 RDN
100 ACX
101 RTN

102♦LBL 06
103 R↑
104 R↑
105 .5
106 FC? 29
107 RND
108 INT
109 3
110 +
111 GTO 04
112 END
 01♦LBL "MP"
 02 SF 08
 03 GTO 00

 04♦LBL "HP"
 05 CF 08

 06♦LBL 00
 07 14
 08 +
 09 E3
 10 /
 11 15
 12 +
 13 STO 11
 14 RCL 12
 15 FC? 04
 16 .123456789
 17 STO 12
 18 FS? 10
 19 GTO 00
 20 FS? 07
 21 GTO 13

 22♦LBL 01
 23 FRC
 24 E1
 25 *
 26 14
 27 +
 28 CLA
 29 XEQ IND X
 30 FRC
 31 E3
 32 *
 33 ACCOL
 34 " "
 35 ARCL IND 11
 36 ACA
 37 PRBUF
 38 RDN
 39 ISG 11
 40 GTO 01
 41 FIX 3
 42 "Y: "
 43 ARCL 00
 44 "├ TO "
 45 ARCL 01
 46 PRA
 47 "X: "
 48 ARCL 08
 49 "├ TO "
 50 ARCL 09
 51 PRA
 52 "ΔX="
 53 ARCL 10
 54 PRA

 55♦LBL 13
 56 XEQ 11
 57♦LBL 00
 58 RCL 02
 59 E
 60 -
 61 RCL 01
 62 RCL 00
 63 -
 64 /
 65 STO 03

 66♦LBL 12
 67 RCL d
 68 55
 69 XROM "IF"
 70 STO ]

 71♦LBL 24
 72 RCL 11
 73 FRC
 74 15
 75 +
 76 STO 11
 77 RCL 12
 78 STO [
 79 24
 80 STO \
 81 FIX 0

 82♦LBL 02
 83 RCL [
 84 FRC
 85 E1
 86 *
 87 14
 88 +
 89 STO [
 90 2
 91 XEQ IND [
 92 FS? 08
 93 GTO 00
 94 INT
 95 RCL X
 96 E5
 97 /
 98 1.007
 99 +
100 STO 05
101 RDN
102 Y↑X
103 STO 13
104 LASTX
105 RCL 10
106 *
107 STO 07
108 E-3

109♦LBL 00
110 FRC
111 STO 06
112 RCL 08
113 STO 14

114♦LBL 03
115 RCL 14
116 RCL IND 11
117 SIGN
118 X=0?
119 GTO 00
120 LASTX
121 GTO 09

122♦LBL 00
123 RDN
124 XEQ IND L

125♦LBL 09
126 RCL 00
127 -
128 RCL 03
129 *
130 E
131 +
132 RND

133♦LBL 25
134 RCL 02
135 X<>Y
136 X<=Y?
137 X<=0?
138 GTO 00
139 GTO 13

140♦LBL 00
141 FS? 05
142 GTO 00
143 X>Y?
144 X<>Y
145 X<=0?
146 E
147 FS? 06
148 CHS
149 GTO 13

150♦LBL 00
151 FC? 06
152 GTO 09
153 X<=Y?
154 GTO 00
155 -
156 RCL 02
157 +

158♦LBL 00
159 X<0?
160 CHS
161 X=0?
162 E
163 GTO 25

164♦LBL 09
165 X<>Y
166 MOD

167♦LBL 13
168 RCL 06
169 +
170 STO IND \
171 E
172 ST+ \
173 FS? 08
174 GTO 04
175 RCL 13
176 ST* 06
177 RCL 07
178 ST+ 14
179 RCL 14
180 RCL 09
181 X<Y?
182 GTO 04
183 ISG 05
184 GTO 03

185♦LBL 04
186 ISG 11
187 GTO 02
188 E
189 ST- \
190 RCL \
191 E3
192 /
193 24
194 +
195 STO 14
196 STO 13
197 ENTER↑
198 ISG Y
199 GTO 05
200 GTO 08

201♦LBL 05
202 CLX
203 -.977
204 RCL IND Y
205 RCL Z
206 INT
207 ST+ Z

208♦LBL 06
209 RDN
210 RCL IND Y
211 X<=Y?
212 GTO 00
213 ISG Z

214♦LBL 14
215 STO IND Z
216 DSE Z
217 DSE Z
218 GTO 06

219♦LBL 00
220 ISG Z

221♦LBL 14
222 RDN
223 STO IND Y
224 RDN
225 ISG Y
226 GTO 05
227 CLX
228 STO 06
229 RCL IND 14
230 ISG 14

231♦LBL 07
232 ENTER↑
233 INT
234 RCL IND 14
235 INT
236 X=Y?
237 GTO 00
238 X<>Y
239 XEQ 10
240 RCL 14
241 STO 13
242 RCL IND X
243 GTO 08

244♦LBL 00
245 FC? 08
246 GTO 00
247 RCL 14
248 STO 13
249 R↑
250 GTO 08

251♦LBL 00
252 RCL Z
253 LASTX
254 X=Y?
255 GTO 08
256 FRC
257 ST+ IND 13
258 LASTX

259♦LBL 08
260 ISG 14
261 GTO 07
262 RCL IND 13
263 XEQ 10
264 RCL ]
265 X<> d
266 STO ]
267 PRBUF
268 FS? 10
269 RTN
270 11
271 FS? 08
272 ST/ X
273 RCL 10
274 *
275 ST+ 08
276 RCL 09
277 RCL 08
278 X>Y?
279 GTO 11
280 RCL ]
281 X<> d
282 STO ]
283 FS? 55
284 GTO 12
285 GTO 24

286♦LBL 11
287 FS? 09
288 XEQ IND 04
289 FS? 09
290 RTN
291 SF 12
292 "-------"
293 ACA
294 ACA
295 PRBUF
296 CF 12
297 RTN

298♦LBL 10
299 X<=0?
300 RTN
301 ENTER↑
302 X<> 06
303 -
304 E
305 -
306 SKPCOL
307 RCL IND 13
308 FRC
309 E3
310 *
311 ACCOL
312 RTN

313♦LBL 15
314 1.127
315 RTN
```

APPENDIX J CONTINUED ON PAGE 377.

RD - RECALL DISPLAY MODE

RD is designed to be used to restore the status of flags 16-55 of register d after SD was used to save them. RD maintains the status of flags 0-15 when restoring the remaining flags. If you should wish to clear flags 0-15, execute RF , then RD .

COMPLETE INSTRUCTIONS FOR RD

1. Insert into X the number of the register that was initialized by SD .
2. XEQ RD to restore flags 16-55.
3. RD saves Y, Z, and T in X, Y, and Z. X is placed in L.

See the SD writeup for some examples. RD does not destroy the flag information when it recalls it, so you can XEQ RD several times if you keep changing display modes and want to go back to the previous mode each time. Normal use of this program will be as a subroutine to end a longer program that has substantially altered the display mode. For example, see PPC ROM routine FD .

Routine Listing For: RD	
69♦LBL "RD"	77 STO \
70 SIGN	78 "⊦*****"
71 ARCL IND L	79 X<> \
72 RDN	80 STO d
73 RCL d	81 RDN
74 STO \	82 CLA
75 "⊦**"	83 RTN
76 X<>]	

LINE BY LINE ANALYSIS OF RD

69	LBL RD	
70	SIGN	Store register number into L
71	ARCL IND L	Place star, flags 16-55 into M
72	RDN	Restore stack
73	RCL d	Place present d into x
74	STO N	Place present d into N
75	"⊦**"	Shift alpha left two bytes by appending two stars
76	X<>O	Remove 1st 2 bytes of d (flags 0-15) into x
77	STO N	N = 5 irrelevant bytes + flags 0-15 M = flags 16-55 + 2 stars
78	"⊦*****"	Shift alpha left five bytes
79	X<>N	New d now in x consisting of present flags 0-15 + old flags 16-55
80	STO d	Insert new d
81	RDN	Restore stack
82	CLA	Clear alpha of garbage
83	RTN	

CONTRIBUTORS HISTORY FOR RD

The first display mode save/recall routines (see *PPC CALCULATOR JOURNAL,* V7N5P8) were apparently written by Leigh Borkman (5218). The first synthetic version of RD (see *PPC CALCULATOR JOURNAL,* V7N7P18) was written by Keith Jarett (4360), as was the final ROM version.

FURTHER ASSISTANCE ON RD

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 20,05 — **RD** — SIZE: 001

Stack Usage:
- 0 T: new d
- 1 Z: T
- 2 Y: Z
- 3 X: Y
- 4 L: X

Flag Usage: FLAGS 00-15 UNCHANGED: FLAGS 16-55 RESTORED TO PRE-SD STATUS.
04:
05:
06:
07:
08:
09:
10:
25:

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED
- 15 e: NOT USED

Display Mode: RESTORED TO PREVIOUS (PRE SD) MODE

Angular Mode: RESTORED TO PREVIOUS (PRE SD) MODE

Unused Subroutine Levels: 6

ΣREG: UNCHANGED

Data Registers:
R00: ONE REGISTER SPECIFIED BY USER.
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: .5 seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: SR
Bytes In RAM: 36
Registers To Copy: 40

Other Comments:

STACK AND ALPHA REGISTER ANALYSIS FOR SD & RD

T	Z	Y	X	L	P	O	N	M	L-#	INSTRUCTION
t	z	y	x	1					45	♦LBL "SD"
			1	x					46	SIGN
1	t	z	y						47	RDN
t	z	y	d						48	RCL d
								1 2 3 4 5 6 7	49	STO [
							1 2 (last two boxes)	3 4 5 6 7	50	"⊢♦♦"
			34567--					1 2 3 4 5 6 7	51	X<> [
									52	GTO 14
										¦
									62	♦LBL 14
					Cleared	Cleared	Cleared	* (last box)	63	"*"
			------*					3 4 5 6 7	64	X<> [
							* (last box)		65	STO \
									66	ASTO IND L
------*	t	z	y						67	RDN
									68	RTN
------*	t	z	y	x			* (last box)	3 4 5 6 7		
t	z	y	x	1					69	♦LBL "RD"
			1	x					70	SIGN
								* 3 4 5 6 7 (from 2nd box)	71	ARCL IND L
1	t	z	y						72	RDN
t	z	y	d'						73	RCL d
							1'2'3'4'5'6'7'		74	STO \
						1'2' (last two boxes)	3'4'5'6'7' * (* in last box)	3 4 5 6 7 * *	75	"⊢**"
			----1'2			1'2'3'4'5'6'7'			76	X<>]
							1'2' (last two boxes)		77	STO \
					1'2'3'4'5' (from 3rd box)	6'7'	1'2'3 4 5 6 7	* * * * * * *	78	"⊢*****"
			d"				1'2' (last two boxes)		79	X<> \
									80	STO d
d	t	z	y						81	RDN
					Cleared	Cleared	Cleared	Cleared	82	CLA
									83	RTN
d	t	z	y	x						

RF - RESET FLAGS

RF sets all flags to their default status, i.e., the state they would be in after MEMORY LOST. The one exception is that RF sets the FIX 2 display mode rather than FIX 4, in accordance with longstanding HP calculator tradition. RF is a short, stand-alone program which can be executed anytime as a cleanup measure via the keyboard, or it can be executed during a running program if it is desirable to have a "virgin" d-register. RF can also be used at the end of any user program to clean up if the program has left several unwanted flags set.

Example 1:

An interesting application of RF program presents itself in the case where one is utilizing a long program and has the printer attached. It is known that having the printer connected to the HP-41 slows down the execution of programs due to the fact that if flag 55 is set, many operations cause the mainframe to check the status of the printer, a time consuming process. If it is not necessary to utilize printer functions in a long program, the speed penalty can be reduced by synthetically clearing flag 55. This clearing must be done during a running program, for as soon as the program halts, the mainframe will sense the existence of the printer and set both flags 55 and 21. Flag 55 can be cleared by: FS? 55 XROM IF. However, XROM RF at the beginning of a program will clear both flags 55 and 21 and they will remain clear until execution halts. RF is also much faster than IF.

COMPLETE INSTRUCTIONS FOR RF

1. XEQ RF

2. The routine ends with the stack undisturbed, with the alpha register clear, and with all flags at their default state (except for FIX 2). If executed from the keyboard with the printer present, the routine will also end with flags 21 and 55 set.

Routine Listing For: RF	
17♦LBL "RF" 18 ",x̄♦" 19 ASTO d	20 CF 03 21 CLA 22 RTN

LINE BY LINE ANALYSIS OF RF

17.	LBL RF	
18.	Hex F4 2C 02 80 00	Inserts flag code into the alpha register. F4 is the text byte; 2C sets flags 26, 28, 29; 02 sets flag 38; 80 sets flag 40.
19.	ASTO d	Transfers new code to register d.
20.	CF 03	Clears flag 3 (set in previous step)
21.	CLA	Eliminates extraneous synthetic code in alpha.
22.	RTN	

CONTRIBUTORS HISTORY FOR RF

This version of RF was devised by Carter Buck (4783) who was probably the first one to discover that the 41C would automatically set flags 21 and 55 if the printer saw that flag 55 was clear.

FURTHER ASSISTANCE ON RF

Call Carter Buck (4783) at (415) 653-6901.
Call Tom Cadwallader (3502) at (406) 727-6869.

TECHNICAL DETAILS		
XROM: 10,13	RF	SIZE: 000

Stack Usage:
- 0 T:
- 1 Z: ALL UNCHANGED
- 2 Y:
- 3 X:
- 4 L:

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: ALTERED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: FLAGS 26, 28, 29, 38 AND 40 ARE SET; THE REST ARE CLEARED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: FIX 2

Angular Mode: DEG

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: .3 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	YES
Program File:	ML
Bytes In RAM:	17
Registers To Copy:	64

Other Comments:

APPENDIX J CONTINUED FROM PAGE 373.

```
316♦LBL 16
317 2.085
318 RTN
319♦LBL 17
320 FS? 08
321 .062
322 FC? 08
323 3.073
324 RTN
325♦LBL 18
326 FS? 08
327 .042
328 FC? 08
329 6.065
330 RTN
331♦LBL 19
332 .028
333 RTN
334♦LBL 20
335 .02
336 RTN
337♦LBL 21
338 8 E-3
339 RTN
340♦LBL 22
341 RCL 33
342 RTN
343♦LBL 23
344 RCL 34
345 END
01♦LBL "BA"
02♦LBL 00
03 "SCAN"
04 CF 21
05 AVIEW
06 XROM 27,05
07 SF 21
08 STO 18
09 FIX 0
10 2
11 X<Y?
12 GTO 00
13 5
14 SF 10
15 GTO 01
16♦LBL 00
17 RCL 02
18 16
19 /
20 INT
21♦LBL 01
22 .
23 X<> d
24 SF IND Y
25 "UT"
26 FS?C 01
27 "PC"
28 FS?C 02
29 "PP"
30 FS?C 04
31 "DE"
32 FS?C 05
33 "PK"
34 FS?C 06
35 "ND"
36 FS?C 07
37 "AR"
38 FS?C 08
39 "AA"
40 FS?C 09
41 "NS"
42 FS?C 10
43 "ARS"
44 FS?C 11
45 "AAS"
46 X<> d
47 ASTO Z
48 "TYPE "
49 ARCL Y
50 "├ "
51 ARCL Z
52 PRA
53 RCL 18
54 CF 29
55 E3
56 /
57 ISG X
58 STO 17
59 CF 12
60 "BY‡    BIN"
61 "├    DEC HEX CH"
62 PRA
63 CF 09
64 CF 08
65 RCL 01
66 CHS
67 STO 00
68♦LBL 02
69 CLA
70 ARCL 17
71 "├ "
72♦LBL 03
73 8
74 RCL IND 17
75 ENTER↑
76 FC? 09
77 ST+ 00
78 128
79 X<=Y?
80 ST- Z
81 X<=Y?
82 SF 08
83 /
84♦LBL 04
85 ENTER↑
86 INT
87 ARCL X
88 -
89 ST+ X
90 DSE Z
91 GTO 04
92 RDN
93 RCL IND 17
94 "├ "
95 E1
96 X>Y?
97 "├ "
98 X↑2
99 X>Y?
100 "├ "
101 ARCL Y
102 "├ "
103 %
104 ACA
105 16
106 MOD
107 X<>Y
108 LASTX
109 /
110 INT
111 XEQ 06
112 X<>Y
113 XEQ 06
114 FS?C 09
115 GTO 05
116 3
117 FC? 08
118 DSE X
119 SKPCHR
120 R↑
121 FC?C 08
122 ACCHR
123 ADV
124 ISG 17
125 GTO 02
126 FS?C 10
127 GTO 05
128 " CHECKSUM:"
129 PRA
130 CLA
131 CLX
132 STO 17
133 SF 09
134 RCL 00
135 X=0?
136 GTO 03
137 255
138 MOD
139 X=0?
140 LASTX
141 STO 00
142 GTO 03
143♦LBL 05
144 3
145 SKPCHR
146 ADV
147 FS?C 07
148 RTN
149 GTO 00
150♦LBL 06
151 9
152 X<Y?
153 GTO 07
154 RDN
155 CLA
156 ARCL X
157 ACA
158 RTN
159♦LBL 07
160 RDN
161 55
162 +
163 ACCHR
164 END
01♦LBL "M2"
02 XEQ 00
03 X<>Y
04♦LBL 01
05 ST* IND Y
06 ISG Y
07 GTO 01
08 RTN
09♦LBL "M3"
10 STO [
11 RDN
12 XEQ 00
13 X<>Y
14 XEQ 00
15 RCL [
16 SIGN
17♦LBL 02
18 RDN
19 RCL IND Y
20 LASTX
21 *
22 ST+ IND Y
23 ISG Y
24 ""
25 ISG Z
26 GTO 02
27 RTN
28♦LBL "M1"
29 XEQ 00
30 X<>Y
31 XEQ 00
32♦LBL "BE"
33 RCL IND Y
34 X<> IND Y
35 STO IND Z
36 RDN
37 ISG X
38 ""
39 ISG Y
40 GTO "BE"
41 RTN
42♦LBL 00
43 RCL 08
44 *
45 RCL 07
46 +
47 RCL X
48 RCL 08
49 ST- Z
50 SIGN
51 -
52 E3
53 /
54 +
55 RTN
56♦LBL "M4"
57 RCL 07
58 -
59 RCL 08
60 XROM "QR"
61 ISG Y
62 ""
63 ISG X
64 ""
65 RTN
66♦LBL "M5"
67 X<> 08
68 ST- 08
69 *
70 ST+ 08
71 X<> L
72 X<> 08
73 1
74 -
75 RCL 07
76 +
77 RTN
78♦LBL 03
79 RCL 07
80 RCL 08
81 RCL Z
82 *
83 +
84 STO Y
85 RCL 09
86 R↑
87 -
88 RCL 08
89 RTN
90♦LBL "IR"
91 ISG 09
92 ""
93 XEQ 03
94 ST- T
95 *
96 GTO "BM"
97♦LBL "DR"
98 XEQ 03
99 ST- Z
100 *
101 DSE 09
102 ""
103♦LBL "BM"
104 SIGN
105 RDN
106 X<Y?
107 GTO 04
108 LASTX
109 ST+ Z
110 +
111 -1
112 ST+ Z
113 ST+ Y
114 RDN
115♦LBL 04
116 R↑
117♦LBL 05
118 RCL IND Z
119 STO IND Z
120 RDN
121 ST+ Z
122 ST+ Y
123 DSE L
124 GTO 05
125 RTN
126♦LBL "BR"
127 CHS
128 X<0?
129 GTO 07
130 RCL Y
131 X<>Y
132 1
133 ST+ Z
134 -
135 SIGN
136♦LBL 06
137 RCL IND Z
138 X<> IND Z
139 STO IND T
140 RDN
141 ST+ Z
142 ST+ Y
143 DSE L
144 GTO 06
145 RTN
146♦LBL 07
147 CHS
148 1
149 -
150 +
151 STO Y
152 -1
153 ST+ Z
154 GTO 06
155♦LBL "BX"
156 STO [
157 STO \
158 STO ]
159 RCL IND X
160 FS? 10
161 ABS
162 ENTER↑
163 ENTER↑
164 RDN
165♦LBL 08
166 CLX
167 RCL IND Z
168 FS? 10
169 ABS
170 X>Y?
171 GTO 10
172 R↑
173 X>Y?
174 GTO 11
175 RDN
176♦LBL 09
177 ISG Z
178 GTO 08
179 X<>Y
180 R↑
181 RTN
182♦LBL 10
183 X<>Y
184 CLX
185 RCL Z
186 STO [
187 GTO 09
188♦LBL 11
189 CLX
190 RCL T
191 STO \
192 X<>Y
193 RDN
194 GTO 09
195♦LBL "BΣ"
196 CLΣ
197♦LBL 12
198 RCL IND Y
199 RCL IND Y
200 Σ+
201 R↑
202 R↑
203 ISG X
204 ""
205 ISG Y
206 GTO 12
207 RTN
208♦LBL "BC"
209 SIGN
210 CLX
211♦LBL 13
212 STO IND L
213 ISG L
214 GTO 13
215 RTN
216♦LBL "UR"
217 1
218 -
219 RCL 10
220 X<>Y
221 Y↑X
222 RCL IND 11
223 X<>Y
224 ST/ Y
225 X<>Y
226 INT
227 RCL 10
228 MOD
229 RTN
230♦LBL "PR"
231 XROM "UR"
232 X<>Y
233 ST* Z
234 *
235 ST- IND 11
236 X<>Y
237 ST+ IND 11
238 RTN
239 END
01♦LBL "S1"
02 X>Y?
03 X<>Y
04 RDN
05 X>Y?
06 X<>Y
07 R↑
08 X>Y?
09 X<>Y
10 R↑
11 X<=Y?
12 GTO 01
13 X<>Y
14 RDN
15 X<=Y?
16 GTO 02
17 X<>Y
18 RDN
19 X<=Y?
20 GTO 03
21 X<>Y
22♦LBL 03
23 RDN
24 RDN
25 GTO 01
26♦LBL 02
27 R↑
28♦LBL 01
29 FS?C 10
30 RTN
31 RDN
32 X<> Z
33 RTN
34♦LBL "S3"
35 STO ]
36 SF 10
37♦LBL 09
38 STO 02
39♦LBL 05
40 STO 01
41 FRC
42 E3
43 *
44 STO \
45 RCL 01
46 INT
47 -
48 E
49 +
50 STO [
51 RCL 01
52 E-4
53 +
54 .
55 RCL X
56♦LBL 14
57 RCL IND Z
58 +
59 ISG Y
60 ""
61 ISG Z
62 GTO 14
63 X<>Y
64 /
65 RCL \
66 X<>Y
67 RCL 01
68 STO T
69♦LBL 13
70 X<> IND \
71♦LBL 11
72 X<Y?
73 GTO 12
74 DSE \
75 ""
76 DSE [
77 GTO 13
78 GTO 06
79♦LBL 12
80 X<> IND T
81 ISG T
82 ""
83 DSE [
84 GTO 11
85♦LBL 06
86 X<> IND Z
87 FRC
88 R↑
89 INT
90 +
91♦LBL 10
92 ENTER↑
93 FRC
94 E3
95 *
96 X<>Y
97 -
98 31
99 X>Y?
100 GTO 01
101 LASTX
102 FS?C 10
103 GTO 09
104 GTO 05
105♦LBL 01
106 LASTX
107 XROM "S2"
108 INT
109 RCL ]
110 INT
111 X=Y?
112 GTO 08
113 RDN
114 RCL 01
115 INT
116 X<>Y
117 X>Y?
118 GTO 07
119 RCL 02
120 INT
121 X<>Y
122 X>Y?
123 GTO 07
124 E
125 -
126 E3
127 /
128 RCL ]
129 INT
130 +
131 SF 10
132 GTO 09
133♦LBL 07
134 E
135 -
136 E3
137 /
138 +
139 GTO 10
140♦LBL 08
141 LASTX
142 BEEP
143 RTN
144♦LBL "S2"
145 CF 10
146 STO [
147 INT
148 E99
149 STO ↑
150 ENTER↑
151 ENTER↑
152 ENTER↑
153♦LBL 04
154 X<> [
155 STO \
156 X<> [
157♦LBL 00
158 X<> IND \
159 X<Y?
160 GTO 01
161 ISG \
162 GTO 00
163 SF 10
164 GTO 02
165♦LBL 01
166 R↑
167 X<Y?
168 GTO 01
169 X<> ↑
170 R↑
171 R↑
172 ISG \
173 GTO 00
174 GTO 02
175♦LBL 01
176 X<> ↑
177 X<Y?
178 GTO 01
179 RDN
180 X<> ↑
181 RDN
182 ISG \
183 GTO 00
184 GTO 02
185♦LBL 01
186 X<> T
187 X<Y?
188 X<>Y
189 R↑
190 X<> ↑
191 X<> T
192 ISG \
193 GTO 00
194♦LBL 02
195 R↑
196 X<> IND [
197 X<> ↑
198 ISG [
199 X<> IND [
200 R↑
201 ISG [
202 X<> IND [
203 R↑
204 ISG [
205 X<> IND [
206 FS?C 10
207 GTO 03
208 R↑
209 ISG [
210 X<> IND [
211♦LBL 03
212 ISG [
213 GTO 04
214 LASTX
215 RTN
216 END
01♦LBL "NS"
02 SIGN
03 RDN
04 ENTER↑
05 "*"
06 X<> [
07 STO \
08 ASTO IND L
09 ASHF
10 ISG L
11 ""
12 ASTO IND L
13 RDN
14 RTN
15♦LBL "NR"
16 RDN
17 ARCL IND T
18 ISG T
19 ""
20 RCL IND T
21 X<> [
22 "├*****"
23 STO \
24 "├**"
25 X<> \
26 RTN
27♦LBL "PO"
28 ADV
29 ADV
30 ADV
31 ADV
32 ADV
33 RTN
34♦LBL "Rb"
35 RCL b
36 RTN
37♦LBL "AM"
38♦LBL 01
39 ASTO IND X
40 ASHF
41 ISG X
42 GTO 01
43 RTN
44♦LBL "MA"
45 CLA
46♦LBL 02
47 ARCL IND X
48 ISG X
49 GTO 02
50 RTN
51 END
01♦LBL "SM"
02 XEQ 04
03 XEQ 04
04 XEQ 04
05 XEQ 04
06 LASTX
07 STO IND 06
08 4
09 ST- 06
10 RTN
11♦LBL 04
12 STO IND 06
13 RDN
14 1
15 ST+ 06
16 RDN
17 RTN
18♦LBL "SE"
19 XROM "RN"
20 RCL 07
21 *
22 RCL 06
23 ST+ Y
24 DSE 07
25 ""
26 RCL 07
27 +
28 RCL IND X
29 X<> IND Z
30 STO IND Y
31 RTN
32♦LBL "XL"
33 X<>Y
34 640
35 +
36 64
37 *
38 +
39 256
40 XROM "QR"
41 X<>Y
42 RTN
43♦LBL "VF"
44 50
45 FC? 50
46 XROM "IF"
47 "βP♦♦♦♦"
48 RCL [
49 SIGN
50 "α          ♦:"
51 X<> [
52 4
53 RDN
54 "FLAGS SET:"
55 X<> d
56 XROM "VA"
57 X<> d
58 CLA
59♦LBL 01
60 FS? IND L
61 XEQ 02
62 ISG L
63 GTO 01
64 X<> d
65 FC?C 24
66 XROM "VA"
67 ADV
68 BEEP
69 X<> d
70 RDN
71 RTN
72♦LBL 02
73 "├ "
74 X<> d
75 ARCL L
76 CF 24
77 DSE T
78 GTO 03
79 XROM "VA"
80 TONE 6
81 CLA
82 4
83 RDN
84 SF 24
85♦LBL 03
86 X<> d
87 END
```

C PPC 1981

ROM 10	ROM 20
MK	SR
1K	Sb
+K	LR
F?	SD
LF	SK
CK	RD
VA	RK
UD	BV
PK	CV
A?	IG
DC	SV
ML	FD
RF	FR
Σ?	DF
S?	NP
C?	GN
DT	RN
AD	BD
XE	TB
HD	PM
ΣC	CM
LB	CJ
L-	JC
-B	CA
XD	LG
VM	HA
EX	HS
MT	CP
DS	MP
VS	HP
EP	BA
TN	M2
CX	M3
CU	M1
CD	BE
VK	M4
AL	M5
NC	IR
SU	DR
NH	BM
HN	BR
BL	BX
FL	BΣ
BI	BC
IP	UR
PS	PR
T1	S1
MS	S3
IF	S2
CB	NS
RT	NR
PD	PO
DP	Rb
QR	AM
2D	MA
SX	SM
RX	SE
OM	XL
PA	VF
GE	
Ab	
E?	
FI	

APPENDIX J CONTINUED ON PAGE 387.

RK - REACTIVATE KEY ASSIGNMENTS

RK is a routine to be used to restore the key assignment bits to their respective registers after **SK** has been used to deactivate them. **RK** can be operated either from the keyboard or in a running program. Once it has been executed, any key assignments that were previously made again become active and take precedence over any local label key functions (A-J, a-e). Don't use **RK** unless **SK** was used first, or you can get some strange machine behavior.

COMPLETE INSTRUCTIONS FOR RK

1. Insert into X the beginning address of the previously stored key assignment bit maps (see step 1 of **SK** instructions).
2. XEQ **RK**
3. User keys are now active.
4. If used in a program, make sure that the correct register number is in X before executing **RK**, and that the registers have not been disturbed since **SK** was executed.
5. **RK** saves Y, Z, and T in X, Y, and Z. x + 1 is place in L.

Routine Listing For: RK	
84♦LBL "RK"	92 X<> \
85 SIGN	93 STO '
86 ARCL IND L	94 X<> [
87 "⊢♦"	95 STO e
88 ISG L	96 RDN
89 ""	97 CLA
90 ARCL IND L	98 RTN
91 "⊢‡"	

LINE BY LINE ANALYSIS OF RK

84	LBL **RK**	
85	SIGN	store register number into L
86	ARCL IND L	alpha recall of first key bit map
87	Hex F2 7F 00	shift alpha left one byte
88	ISG L	increment L to obtain second address
89	Hex F0	NOP
90	ARCL IND L	alpha recall of second key bit map
91	Hex F3 7F 0F FF	append 0F FF to alpha
92	X<>N	X now contains 1st 5 bytes of ⊦ + null + star
93	STO ⊦	restore unshifted key bits
94	X<>M	X now contains 1st 5 bytes of e + 0F FF
95	STO e	restore shifted key bits
96	RDN	restore stack
97	CLA	clear alpha of garbage
98	RTN	

NOTE: The purpose of the FFF inserted into alpha at line 91 and then into register e at line 95 is to insure that the calculator mainframe recomputes the program line number when it is next switched to program mode. It also insures the correct line number if the program is single stepped or run in trace mode (with printer).

CONTRIBUTORS HISTORY FOR RK

The first version of **RK** (see *PPC CALCULATOR JOURNAL*, V7N7P18) was written by Keith Jarett (4360) in response to a suggestion by Gary Tenzer (1816). The ROM version of **RK** was written by Roger Hill (4940).

FURTHER ASSISTANCE ON RK

Call Keith Jarett (4360) at (213) 374-2583.
Call Keith Kendall (5425) at (801) 967-8080.

TECHNICAL DETAILS

XROM: 20,06 | **RK** | SIZE: 002

Stack Usage:
- 0 T: new c
- 1 Z: T
- 2 Y: Z
- 3 X: Y
- 4 L: X + 1

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: CHANGED (RESTORED)
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: NOT USED
- 15 e: CHANGED (RESTORED)

ΣREG: UNCHANGED

Data Registers:
R00: NO REGISTERS ARE ALTERED

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: .6 seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: **SR**
Bytes In RAM: 32
Registers To Copy: 40

Other Comments:

RK

STACK AND ALPHA REGISTER ANALYSIS FOR SK & RK

T	Z	Y	X	L	P	O	N	M	L-# INSTRUCTION
t	z	y	x	1					53♦LBL "SK"
			1	x					54 SIGN
			0						55 CLX
			⊦						56 X<> ↑
									57 XEQ 14
									62♦LBL 14
					Cleared	Cleared	Cleared	*	63 "*"
			-----*					1 2 3 4 5 6 7	64 X<> [
							*		65 STO \
									66 ASTO IND L
-----*	t	z	y						67 RDN
									68 RTN
				X+1					58 ISG L
									59 ""
t	z	y	0						60 .
			e						61 X<> e
									62♦LBL 14
					Cleared	Cleared	Cleared	*	63 "*"
			-----*					8 9 10 11 12 13 14	64 X<> [
							*		65 STO \
									66 ASTO IND L
-----*	t	z	y						67 RDN
									68 RTN
-----*	t	z	y	x+1			*	8 9 10 11 12 13 14	
t	z	y	x	1					84♦LBL "RK"
			1	x					85 SIGN
								* 1 2 3 4 5	86 ARCL IND L
								* 1 2 3 4 5	87 "⊦•"
				X+1					88 ISG L
							* 1 2 3 4 5	* 8 9 10 11 12	89 ""
									90 ARCL IND L
						*	1 2 3 4 5 *	8 9 10 11 12 OFFF	91 "⊦‡"
			⊦'				*		92 X<> \
									93 STO ↑
			e'					1 2 3 4 5 *	94 X<> [
									95 STO e
e'	t	z	y						96 RDN
					Cleared	Cleared	Cleared	Cleared	97 CLA
e'	t	z	y	x+1					98 RTN

RN - RANDOM NUMBER GENERATOR

This routine is a random number generator which will generate uniformly distributed pseudo-random numbers r in the range 0<r<1. The resulting random numbers can be re-scaled to produce uniform numbers within any specified range. Input to this routine requires a register pointer value which points to the register which will hold the starting seed as well as the subsequent random number decimals. RN will produce a million distinct random decimals before cycling, regardless of the initial seed.

Example 1: Use RN to produce a series of random numbers r within the range 0<r<1.

This routine requires the use of one data register. There is no restriction on which register is used. For this example we will use R05. Before RN can be called for the very first time we must store a random decimal between 0 and 1 in our chosen data register, in this case R05. For the purposes of this example we will use the fractional part of pi as the initial seed. Key pi XEQ "FRC" STO 05 (.141592654).

Do:	See:	Result:
5 XEQ "RN"	0.792782000	1st random number
5 XEQ "RN"	0.123349000	2nd random number
5 XEQ "RN"	0.621856000	3rd random number
5 XEQ "RN"	0.459103000	4th random number

Any number of subsequent random decimals in the range 0<r<1 can be generated by simply keying in 5 and XEQ "RN". The initial decimal stored is called the starting seed. The initialization need be performed only once to produce thousands of random numbers.

COMPLETE INSTRUCTIONS FOR RN

1. One data register is required to hold the seeds. The number (address) of this data register is the input to RN. Call this register k.

2. Before the first call to RN store any decimal between 0 and 1 in the designated data register. (register k).

3. To generate a random decimal r, 0<r<1, key k in X and XEQ "RN". r will be left in X as well as in register k.

4. Step 3. may be repeated any number of times.

5. The stack register contents on input/output are indicated by:

	Input:	Output:
T:	T	Y
Z:	Z	Y
Y:	Y	k (pointer)
X:	k (pointer)	r
L:	L	*

MORE EXAMPLES OF RN

Example 2: In almost all applications RN will be used as an internal subroutine. To generate random numbers in the range between the limits a and b where a<b the following formula may be used.

$$x = r(b-a) + a$$

where r is a random decimal produced by RN (0<r<1) and x represents a random decimal where a < x < b.

More specifically, if RN were used to roll a single die we would want to generate a random integer in the range between 1 and 6 inclusive. In this case take a=1 and b=7. The following routine which calls RN will leave an integer between 1 and 6 inclusive in X. As in Example 1 we arbitrarly choose register R05 to hold the seeds.

```
LBL "DIEROLL"
    5       (register pointer)
 XROM RN    (produce decimal 0-1)
    6       ( 6 = b-a = 7-1 )
    *       ( r*(b-a) )
    1       ( a=1 )
    +       ( x = r(b-a) + a )
   INT      ( die = INT(x) )
   RTN
```

Example 3: Now suppose you are dealing cards from a standard deck of 52 cards. Assume further that the card values are stored in registers R21-R72 inclusive. In this case to deal a random card will require a random integer between 21 and 72. See Example 2. Take a=21 and b=73. The following routine which calls RN will leave the card value in X. Again we use register R05 to hold the seeds.

```
LBL "CARDEAL"
    5        (register pointer)
 XROM RN     (produce decimal 0-1)
   52        ( 52 = b-a = 73-21 )
    *        ( r(b-a) )
   21        ( a=21 )
    +        ( x = r(b-a) + a )
 RCL IND X   ( use INT part of x )
   RTN
```

(NOTE: For dealing cards see also the selection without replacement routine SE.)

FORMULAS USED IN RN

Only one formula is used in RN.

$S_{i+1} = FRC(9821*S_i + 0.211327)$

Routine Listing For: RN	
145♦LBL b 146♦LBL "RN" 147 RCL IND X 148 9821 149 * 150 .211327 151 + 152 FRC 153 STO IND Y 154 END	

LINE BY LINE ANALYSIS OF RN

Lines 145-154 are the only lines used in the short RN routine. The original pointer input is left in Y and is used a second time at line 153.

REFERENCES FOR RN

1. Vic Heyman (850) 65 NOTES "Random Number Generators" V4N8P1-8 (This is a must read article)

2. Donald Knuth, "Semi-Numerical Algorithms," Volume 2 The Art of Computer Programming, Addison Wesley, 1969 (Section 3.4)

CONTRIBUTORS HISTORY FOR RN

The subject of RNG's is an important one. The method chosen here is an HP-65 routine by Don Malm (1362) and was listed in the HP-34C Applications Booklet p. 57 and the HP-41C Standard Applications Booklet p. 24. John Kennedy (918) wrote the documentation for RN.

FURTHER ASSISTANCE ON RN

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

TECHNICAL DETAILS

XROM: 20, 16 — RN — SIZE: 001 minimum

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:

R00: RN uses one data register specified by the user
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine: b

Execution Time: 1 second

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: FR

Bytes In RAM: 29

Registers To Copy: 36

Other Comments:

RT - RETURN ADDRESS TO DECIMAL

RT decodes the first return in the subroutine return stack, provided that the return address is to a point in RAM (not to a point in a ROM). The routine starts with the register b contents previously recalled into X, and ends with X containing the integer return address in absolute bytes. **RT** uses and clears the alpha register, and uses the stack, saving only the Y register contents.

Example 1: This artificial example of manual use of **RT** illustrates its operation. Enter the program lines as follows.

```
01  LBL "RTT"
02  "OK"            06  LBL 01
03  XEQ 01          07  PROMPT
04  "DONE"          08  RTN
05  PROMPT
```

Assume a key has been assigned the synthetic function RCL b. (Refer to the **MK** routine.)

DO:	SEE:	RESULT:
XEQ"RTT"	"OK"	Program stopped after line 07 PROMPT in subroutine called by line 03 XEQ 01
RCL b	an NNN	b register contents copied into X
XEQ **RT**	a decimal integer	Return address in absolute bytes

The result is the byte address of the last byte in the line 03 XEQ 01, which is the line which called the subroutine LBL 01. That also is the program pointer value when, in program mode, one sees the display of the next line, 04 "DONE".

COMPLETE INSTRUCTIONS FOR RT

RT operates on a copy of the b register contents which was previously placed in the X register (usually by a RCL b). It ends with the X register containing an integer (a decimal number) which is the byte number address for the pending subroutine return. Y, Z, and T registers contain the original contents of Y, and L contains 7 times the absolute address of the register in which the return address byte lies. The alpha registers were used and cleared.

The use of **RT** consists of two steps:

a) Copy the b register into X at a point when the desired address is the pending subroutine return.

b) Execute **RT**

However, the uses to which **RT** will be put may all be quite complicated ones, such as programs which write program lines in themselves or in associated programs. Therefore, this description necessarily falls far short of giving a complete account of how to use **RT** .

Formats of Return Address and Program Pointer:

The program pointer and the subroutine return address stack (up to 6 return addresses) are in the a and b registers. The two rightmost bytes of b (bytes 1 and 0) contain the program pointer, the next two bytes (bytes 3 and 2) contain the pending subroutine return address, and so on through the rest of the b and then the a register.

First, consider the program pointer format. As shown in the diagram, the three right nybbles (12 bits)

```
etc|  Pending Return        |  Program Pointer       |
 <-       Address           |                        |
   |Byte 3    |Byte 2       |Byte 1    |Byte 0       |
   |0000| BBBR|RRRR| RRRR   |0BBB|000R |RRRR RRRR    |

     Byte       Register  Byte       Register
      in        Number     in        Number
     Register             Register
```

of the program pointer contain the register number in binary form, and the left nybble (4 bits) contains the byte number within the register. The byte number runs from 0 to 6, so, as shown, only three bits are used, and the leftmost bit is always zero. Since the absolute register numbers can at most extend from 0 to 15 for status registers and from 192 to 511 with maximum memory, only 9 bits of the 12 available for register address are needed, and the left three are always zero for addresses in RAM. (When the program pointer is in a ROM, all 16 bits are used, in a simpler format. See **Ab** .)

When a subroutine is called by an XEQ line, the address of the last byte in the XEQ instruction is put into the subroutine stack, bytes 3 and 2 of register b. However, the format is different from that of the program pointer for addresses in RAM. As shown in the diagram, the program pointer format is converted to the return address format by shifting the 3 bits of the byte number (byte number within the register) into the space of the always-zero bits in the register number. Thus, the entire left nybble is always zero for return addresses in RAM. When the return address is to a point in a ROM, the left nybble is not zero, and the format is the same as when the program pointer is in a ROM. This fact is used in the **XE** (XROM Entry) routine.

MORE EXAMPLES OF RT

Example 2: The **LB** routine, complex though it is, has to be the best real example of a use for **RT** , since that is why **RT** is in the PPC ROM. Briefly, **LB** must have a reference address in the user's program in order to load bytes into it. **LB** gets that address by getting called as a subroutine by a temporary line in the user's program, then recalling the b register contents (which contain the return address) into X, then finally executing **RT** to decode the return address.

Routine Listing For:	RT
40♦LBL "RT" 41 STO [42 "├*****" 43 X<> \ 44 XROM "2D" 45 2 46 / 47 INT 48 LASTX 49 FRC 50 512 51 GTO 14	58♦LBL 14 59 * 60 RCL [61 + 62 7 63 * 64 + 65 CLA 66 RTN

LINE BY LINE ANALYSIS OF RT

The operation of RT is clarified by referring to the format repeated here for the return address in bytes 3 and 2 of the b register.

Byte 3		Byte 2	
0000	BBBR	RRRR	RRRR
Always zero	3 bits byte number	9 bits register number	

the desired result is the byte number plus 7 times the register number.

At the start, X contains the contents read from the b register.

Lines 41-43 place the two bytes of interest (original bytes 3 and 2 from b, shown above) into the two right-most bytes of X.

Line 44 calls the 2D routine, decoding the two bytes separately into two decimal numbers: Byte 2 is in M and Byte 3 is in X.

Lines 45-49 slice the number in X (Byte 3) into two numbers: X then contains half the value of the least significant bit (the R bit) and Y contains the (once right-shifted) value of the three B bits.

Lines 50-51 and 58-61 combine the nine R bits, getting the assembled decimal register absolute address.

Lines 62-66 assemble the absolute byte address.

REFERENCES FOR RT

See *PPC CALCULATOR JOURNAL*, V7N10P20, 'First Byte Loader Program'.

See Routine LB for application using RT.

CONTRIBUTORS HISTORY FOR RT

Keith Jarett (4360) wrote the first RT based on Roger Hill's (4940) idea. Roger Hill wrote the final RT program.

FURTHER ASSISTANCE ON RT

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,51 — **RT** — SIZE: 000

Stack Usage:
0 T: Y
1 Z: Y
2 Y: Y
3 X: result
4 L: 7* reg. number

Alpha Register Usage:
5 M:
6 N: ALL CLEARED
7 O:
8 P:

Other Status Registers: NONE USED
9 Q:
10 ⊢:
11 a:
12 b:
13 c:
14 d:
15 e:

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
2D	NONE

Local Labels In This Routine:
NONE

Execution Time: 1.7 seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: IF
Bytes In RAM: 29
Registers To Copy: 60

Other Comments:

RX - RECALL FROM ABSOLUTE ADDRESS IN X

RX can be used to recall data, program bytes or key assignments from any desired register in user memory. For a map of user memory, see **LF** Figure 1.

RX is helpful when data is stored (using **SX**) in the unused memory space between the .END. and the key assignment registers. When **RX** is used to recall program bytes or key assignments, there is an undesirable consequence--the chosen register is normalized. If the first byte of the register is null, all 7 bytes will be cleared. In the case of key assignments, the leading F0 byte is changed to 10. This normalization applies to both the register contents and the recalled copy. Don't use **RX** to recall program bytes or key assignments unless you really know what you're doing. For instance, if you fail to put back the F0 leading byte of a key assignment register, all key assignments in or above that register are "lost".

Example 1: Recall data register 05 using **RX**.
Do π, STO 05, CLX, XEQ **C?**, 5, +, XEQ **RX**. You should get π back.

Example 2: Continuing the above example, recall data from below the curtain. Do 6 XEQ **CU**. This puts R05 just below the curtain. Now do XEQ **C?** 1 - XEQ **RX**. You should get π again. Do -6 XEQ CU to get back the original curtain.

COMPLETE INSTRUCTIONS FOR RX

RX is usually used to recall data stored by **SX**. **RX** can also be used to access data which is temporarily below the curtain. To use **RX**, just place the chosen decimal absolute address in X and XEQ **RX** to recall the chosen register. X must be at least 192 and less than 256 + n*64, where n is the number of (single density) RAM modules present.

The stack usage of **RX** is as shown.

	Before		After
T	don't care	T	temporary c register (from **OM**)
Z	z	Z	z
Y	y	Y	y
X	absolute address	X	normalized contents of absolute address
L	don't care	L	absolute address - 16
alpha	don't care	alpha	cleared

If **RX** comes back with something weird-looking in X, you have probably recalled part of your program or key assignments. If you're lucky you haven't destroyed the CAT 1 linkage. Then you can clean up the several lines of program memory affected or use **CK** to clear out disrupted key assignments. If CAT 1 is disrupted, try GTO.. or PACK. If these don't work, chalk it up to experience and MASTER CLEAR.

Note that **RX** leaves the temporary c register (see **OM**) in T. It displays as ⊠ 𐀀 ⊠ 𐀀 x y where x can be 𐀀 or ⁻ and y can be any character. This temporary c register may be useful for subsequent processing when **RX** is used in a program.

If you get NONEXISTENT when using **RX**, you are just one false step away from MEMORY LOST. Press ENTER↑ to raise the stack, switch to PRGM mode, SST, back to RUN mode, and R/S. Instead of getting a recalled number in x you'll have the old c register contents (that you ENTER↑ed).

WARNING - Avoid executing **RX** with flag 25 set. If flag 25 is set and an illegal address is specified, you will get MEMORY LOST. This problem was anticipated when **RX** was written, but there wasn't an easy, foolproof way to eliminate it.

Routine Listing For: **RX**

```
129♦LBL "RX"
130 XEQ 14
131 RCL IND L

132♦LBL 13
133 X<>Y
134 X<> c
135 RDN
136 RTN

137♦LBL 14
138 16
139 -
140 SIGN
141 RDN

142♦LBL "OM"
143 XEQ 14
144 "▓ꜛiꜛ"
145 X<> [
146 STO \
147 "├**"
148 X<> \
149 CLA
150 X<> c
151 RTN

183♦LBL 14
184 RCL c
185 STO [
186 "├♦♦♦♦♦"
187 X<> [
188 X<> d
189 CF 00
190 CF 01
191 CF 02
192 CF 03
193 X<> d
194 RTN
```

LINE BY LINE ANALYSIS OF RX

Line 130 executes the curtain lowering routine **OM** and stores X-16 in last x. Line 131 recalls the chosen register, and lines 132-136 restore the curtain and the stack (except T).

CONTRIBUTORS HISTORY FOR RX

RX is a substitute for Bug 2, allowing any user memory register to be recalled. The concept of resetting the curtain location to simulate Bug 2 originated with Bill Wickes' (3735) B2 program (see *PPC CALCULATOR JOURNAL*, V7N3P7a). **RX** is a simple application program for **OM**, and was written by Keith Jarett (4360).

FURTHER ASSISTANCE ON RX

Call Keith Jarett (4360) at (213) 374-2583.
Call Roger Hill (4940) at (618) 656-8825.

RX

TECHNICAL DETAILS

XROM: 10,57 | **RX** | SIZE: 000

Stack Usage:
- 0 T: temporary c
- 1 Z: Z
- 2 Y: Y
- 3 X: result
- 4 L: X-16

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED BUT RESTORED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: SEVERAL USED BUT ALL RESTORED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
OM	PART OF GE

Local Labels In This Routine:
13
14

Execution Time: 1.1 second.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO!
Program File: IF
Bytes In RAM: 54
Registers To Copy: 60

Other Comments:
Avoid using RX with Flag 25 set; this plus an invalid X gives MEMORY LOST.

If you get NONEXISTENT, ENTER↑, SST past line 131 and R/S to restore the curtain.

NOTES

Rb - RECALL b

Rb provides a method to determine which port the PPC ROM is plugged into. XEQ Rb produces a non-normalized number in X that decodes (XEQ NH) to 00 00 00 00 00 WF 12, where W = 9, B, D, or F according as the PPC ROM is in port 1, 2, 3, or 4.

Example 1: The following routine quickly determines which port contains the PPC ROM.

```
LBL"P?"
XROM Rb
XROM 2D
SQRT
11
-
INT
RTN
```

APPLICATION PROGRAM 1 FOR Rb

See the Ab write up for an application program "PE" (PPC ROM entry) which attaches the correct port code to a ROM entry address. Together with XE, this provides a port-independent PPC ROM entry capability.

Routine Listing For: Rb	
34♦LBL "Rb" 35 RCL b 36 RTN	

LINE BY LINE ANALYSIS OF Rb

34 LBL Rb	
35 RCL b	Place the contents of Register b in x.
36 RTN	The last 4 nybbles will be 9F12, BF12, DF12, or FF12.

CONTRIBUTORS HISTORY FOR Rb

Roger Hill (4940) conceived of Rb late in the ROM development. It was added to the PPC ROM when a few bytes became available. The port finder routine was written by Roger Hill and Richard Nelson (1).

FURTHER ASSISTANCE ON Rb

Call Tom Cadwallader (3502) at (406) 727-6869.
Call Roger Hill (4940) at (618) 656-8825.

NOTES

TECHNICAL DETAILS

XROM: 20,52 | **Rb** | SIZE: 000

Stack Usage:
- 0 T: Z
- 1 Z: Y
- 2 Y: X
- 3 X: hex(0000000000WF12
- 4 L: L (W = 7+2*PORT)

Alpha Register Usage:
- 5 M:
- 6 N: ALL UNCHANGED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q:
- 10 ⊢:
- 11 a: NONE USED
- 12 b:
- 13 c:
- 14 d:
- 15 e:

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: Less than .1 second.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? YES
Program File: NS
Bytes In RAM: 9
Registers To Copy: 16

Other Comments:

Rb

APPENDIX J - PPC CUSTOM ROM LISTING

APPENDIX J CONTINUED FROM PAGE 377.

PPC ROM FILE (GROUP) BRANCHING TABLE

(See page 401 for column headings)

MK GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
+K			11001		MK		
1K			006	VS			
F?			008				C P
MK	01		009				
----			010	LF			
01		07	018				C
02		13	021				C
03	02		022				
07			024			11	
08			030	VA			
09	03		033				
11			036	VA			
13			038				S
14-6		14	039				
			040		1K		
			043		+K		
	14		044				
		01	055				C
	13		056				
		02	062				C
		08	089				C
		08	12102				C
		14	108				C
	14		112				
		09	115				C
		14	119				C
	14		123				
			134	DC			
			135	DC			
			136	DC			
		14	138				C
	14		142				
			156				C R
		03	158				C
		14	161				
	07		162				
	14		166				
			168				C R
			169	VA			
			172				S
		01	173				
	08		174				
		14	176				
	09		177				
	14		13180				
			184				C R
			185	VA			
			191	VA			
			193				S
		01	195				
			196		F?		
			197	LF			
	11		198				
			210				E

LF GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
A?			13001		LF		
CK			002	E?			
DC			009	OM			
LF		14	014				C
PK	00		015				
UD		14	018				C
VA		00	027				C
----	14		028				
00			039				R
01			040		CK		
02			041	E?			
03			049				C R
04			050	OM			
05	06		054				
06		06	057				C
07			14061				R
08			062		VA		
14-2			067				A
			070				R
			071		UD		
			076				R
			077		PK		
			078	E?			
			090	OM			
		03	092				
	07		093				
		03	097				C
		08	100				C
		03	106				
	08		107				
		02	112				
	01		113				
		04	118				C
		07	138				C
	02		143				
	03		15147				C
		01	149				C
	04		150				
		05	152				C
	05		157				
		14	162				
			163		A?		
	14		164	LF			
			165				
			174				R
			175		DC		
			208				E

ML GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
AD			15001		ML		
C?			003			13	
DT			011	DC			
HD			016	GE			
ML			017		RF		
RF			022				R
S?			023		Σ?		
XE			025	C?			
Σ?			16027			14	
ΣC			031				R
----			032		S?		
02			033	C?			
13	13		035				
14-2	02		039				
			042				C R
		02	045				
			046		C?		
	14		048				
			076				R
			077		DT		
			092				A
			093				S
			097				R
			098		AD		
			17118				R
			119		XE		
			120			14	
	14		121				
			131				C R
			142				b
			143		HD		
			154		ΣC		
			199				E

NS GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
AM			59001		NS		
MA			014				R
PO			015		NR		
NR			026				R
NS			027		PO		
Rb			033				R
----			034		Rb		
01			036				R
02			037		AM		
	01		038				
		01	042				C
			043				R
			044		MA		
	02		046				
		02	049				C
			050				R
			051				E

LB GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
-B	00		17001				
L-			002				S
LB		++	003				RAM
XD			004		LB		
----		00	18006				C
00-2			008	VA			
01		13	010				
03			011		L-		
06			013	VA			
07		14	019				C
08	13		022				
09			024	VS			
10			026				C P
11		++	033				RAM
12	00		034				
13-2		14	037				C
14-5			040				a
15			042				b
19	14		043				
20			044	RT			
			049	QR			
			056	QR			
		10	068				C
			074	OM			
	06		078				
	07		081				
	15		085				C
			088				C R
			19098	VA			
			100				S
		14	102				C
		19	104				C
		08	105				
	14		106				
		19	108				C
			109	XD			
	08		110				
		03	112				C
			116	DC			
		10	119				C
		07	123				C
		06	135				C
	20		137				C
			150	VA			
		00	152				
	01		153				
		09	156				C
	10		157				
			159	VA			
		00	161				
	03		162				
			164	VA			
		01	167				C
		14	169				C
		10	171				C
	14		172				
		14	178				C

CONTINUED NEXT COLUMN

LB GROUP CONTINUED

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
		13	19181				C
		09	10185				
	13		186				
		15	191				
	14		192				
	09		200				
		15	202				
			203		-B		
		15	205				C
		08	207				C
	19		208				
		10	211				C
		14	217				C
	11		220				
		11	223				C
	14		225				
	12		231				
		12	235				C
		20	239				
			240		XD		
			244	QR			
			257				E

SR GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
BV			34001		SR		
LR			009				C R
RD			021				a
RK			024		Sb		
Sb			025				b
SD			026		LR		
SK			044				R
SR			045		SD		
----		14	052				
00			053		SK		
01			057			14	
14	14		062				
			068				R
			069		RD		
			083				R
			35084		RK		
			098				R
			099		BV		
	00		102				
		01	106				C
			118	VA			
			120				S
	01		126				
		00	129				C
			131				E

APPENDIX J CONTINUED ON PAGE 401.

S1 - STACK SORT

S1 will sort the stack (X, Y, Z, & T) register values (numeric only) into ascending or descending sequence. Flag 10 controls the sequence. Last X is untouched. If alpha data is in the stack the HP-41 will stop with ALPHA DATA in the display.

Example 1: Jack wants to sort the final bowling scores of his four friends. He places the data into the stack and executes S1 . Here is his result.

Data at Start		FLAG 10 SET Ascending	FLAG 10 CLEAR Descending
T	283	295	283
Z	295	291	289
Y	291	289	291
X	289	283	295

COMPLETE INSTRUCTIONS FOR S1

1. Set or clear flag 10. SF 10 will place the lowest value into the X register, the highest into T. CF 10 will invert this sequence placing the highest value into X and the lowest into T.
2. Load the stack with numbers to be sorted.
3. XEQ S1
4. Flag 10 will be cleared at the end of the routine.

MORE EXAMPLES OF S1

Example 2: The speed of sorting programs is data dependent. Henry decided to test the PPC ROM Routine S1 to verify this. He wrote the simple RAM program shown below to time the execution of S1 . The PSE at the beginning allowed a slow, relaxed signal (tone 89) for starting the stopwatch.

01	PSE	Using stack data: T thru X
02	TONE 89	
03	SF 10	6868, 3533, 3379, 7642
04	XROM S1	The run time was 8.51 seconds.
05	XROM S1	
:	lines 03-05 repeated 9 more times	Using a clear stack the time was 6.30 seconds.
33	TONE 89	When none of the routine comparison intructions are "true", the routine runs faster due to fewer instructions to execute. The all zero data verifies this.
34	END	

FURTHER DISCUSSION OF S1

One application for a stack sort would be as part of a full blown data register sort. Sorts done on large mainframes are usually done by creating sorted 'strings' of data, then merging all the strings to create the final sorted file. The stack sort could be used to create 'strings' of four sorted registers which could then be merged by another routine to produce the final sorted file.

Another possible application is in biorythm print programs. Most of them seem to take a long time to compute and print each days values. Possibly using a quick sort to place the three values in order for printing would speed it up. The values to be sorted might have a format such as: XX.YY where XX is the position from -1 to +1 on the horizontal axis and YY is the character (or register holding the character) to be printed there. After determining the starting points for each point, each new point would be caluculated, sorted and formatted onto the printline.

Programming Technique:

Lines 31 and 32 provide a simple means of inverting the stack.

Routine Listing For: S1

```
01♦LBL "S1"        17 X<>Y
02 X>Y?            18 RDN
03 X<>Y            19 X<=Y?
04 RDN             20 GTO 03
05 X>Y?            21 X<>Y
06 X<>Y            22♦LBL 03
07 R↑              23 RDN
08 X>Y?            24 RDN
09 X<>Y            25 GTO 01
10 R↑              26♦LBL 02
11 X<=Y?           27 R↑
12 GTO 01          28♦LBL 01
13 X<>Y            29 FS?C 10
14 RDN             30 RTN
15 X<=Y?           31 RDN
16 GTO 02          32 X<> Z
                   33 RTN
```

LINE BY LINE ANALYSIS OF S1

Ascending order as used here means lowest in X, highest in T. Lines 01 thru 03 place X & Y in ascending order. The roll down in line 04 brings a "new" value into Y. The new Y and old Y are placed into ascending order by lines 05 and 06. Line 07 brings back the old X completing the ascending sort of original X, Y, & Z. The roll up at line 10 brings the T register, not used until this point, into X. The T value is tested at line 11. If the old value of T is less than the previous X value, the sort is complete and the flag 10 test routine 01 is executed in lines 28 thru 33. If the T value is not the lowest value of the stack, lines 13 thru 21 are executed to determine the relative position of this value with respect to the previously sorted values.

Labels 02 and 03 provide convenient entry points depending on the value of T. This approach provides the fastest execution time for sorting. Label 01 is executed every time S1 is called. The routine is essentially complete at line 28 with the

TECHNICAL DETAILS

XROM: 20,46	S1	SIZE: 000

Stack Usage:	Flag Usage:
0 T: USED	04:
1 Z: USED	05:
2 Y: USED	06:
3 X: USED	07:
4 L: NOT USED	08:
Alpha Register Usage:	09:
5 M: NOT USED	10: Lowest value in X if * set.
6 N: NOT USED	
7 O: NOT USED	
8 P: NOT USED	25:
Other Status Registers:	Display Mode: N/A
9 Q:	
10 ⊢:	
11 a: NONE USED BY	Angular Mode: N/A
12 b: ROUTINE	
13 c:	
14 d:	Unused Subroutine Levels:
15 e:	5
ΣREG: NOT USED	Global Labels Called:
Data Registers:	Direct: NONE — Secondary: NONE
R00:	
R06: NONE USED BY	
R07: ROUTINE	
R08:	
R09:	
R10:	
R11:	Local Labels In This Routine:
R12:	LBL 01
	LBL 02
	LBL 03

Execution Time: < 1 second. (approximately 0.4 seconds)

Peripherals Required: NONE

		Other Comments:
Interruptible?	YES	* Flag 10 is cleared by the routine and must be set before execution if lowest value is to be in X.
Execute Anytime?	YES	
Program File:	S1	
Bytes In RAM:	46	
Registers To Copy:	47	

stack in ascending order. Flag 10 is tested and if set, execution stops at the RTN at line 30. If flag 10 is clear the stack order is inverted by lines 31 and 32 with termination at line 33, the last line of the routine.

CONTRIBUTORS HISTORY FOR S1

Harvey C. Meyers (3101) wrote S1 as an ascending stack sort. Richard Nelson (1) added the flag 10 test to add a descending capability. Harvey also provided the rough draft of this documentation.

FINAL REMARKS FOR S1

Large data bases must often be sorted in segments to be later merged. Could S1 be used as a training exercise in writing such routines? Is this the fastest stack sort possible? What order of the data is the fastest, the slowest?

FURTHER ASSISTANCE ON S1

Harvey C. Meyers (3101) -- (312) 665-6681 eve.
Richard J. Nelson (1) -- (714) 754-6226 P.M.

NOTES

S2 - SMALL ARRAY SORT (≤32)

S2 is an expansion of a sorting technique known as simple selection. In simple selection, multiple passes are made through an array. In each pass, a single data (number) with the lowest value is found and exchanged with the number at the lowest address in the remaining unsorted portion of the array. In an array of data with a size of N, N—1 passes are therefore required for completion.

S2 operates in the same manner but selects four data of lowest value in each pass. This reduces the number of passes required to approximately N/4 and consequently reduces execution time, although not by a factor of 4.

To accomplish the selection of four data with each pass, an insertion technique is required to arrange and keep the data in its proper order in the stack. At any intermediate point in a pass through the array the four numbers currently found to be of lowest value are arranged and stored in registers Y, Z, T and P. If these numbers are represented by a, b, c and d, with a the smallest and d the largest, then the arrangement is:

```
P  b
T  a
Z  c
Y  d
X  xn
```

When a new number is placed in X, it is compared with d. If equal to or larger than d the routine immediately replaces it with the next number in the array. If smaller than d, it is inserted in its proper location in Y, Z, T or P and d is placed in X for replacement. When the pass is complete, the four numbers selected are stored in the lowest 4 addresses in the unsorted portion of the array.

S2 will, under certain conditions, select five numbers for storage. This will occur if the last number to be compared with d at the completion of a pass is found to be smaller than d. This, of course, means that the last number and d are members of the five lowest values in the array since no further comparisons will be made during the pass. F10 checks for this condition. In arrays of random data, this may occur more frequently than might be expected since S2 has the somewhat self defeating property of arranging the unsorted portion of a random array in quasi "upside down" order as sorting progresses.

When sorting an inverted array, this condition will occur for each pass and explains why S2 will generally sort such an array slightly faster than a random array, even though it is executing more instructions in the insertion sequence.

S2 was designed for maximum speed as can be seen from the multiple ISG N, GTO 00 instructions which could easily be replaced by a single GTO NN, where the NN label would have performed these steps. In this regard, it should be noted that the LBL 00 loop will be executed approximately 500 times in a random array of 64 numbers, requiring an extra 500 instruction executions if the GTO NN were incorporated.

S2 uses a combination of selection and insertion sorting techniques to provide fast sorting of numerical data arrays. The array to be sorted is addressed by placing a number of the form bbb.eeeii in stack register X, where bbb is the beginning data storage register address, eee is the ending data storage register address and ii is the increment. Beginning register bbb may be any storage register including R00. When sorting is complete, bbb.eeeii is returned to X and the data is sorted in ascending order in the addressed registers.

S2 is capable of sorting arrays of any size although the companion routine S3 will execute much faster for large contiguous arrays. S2 should therefore be limited to arrays <32 or where data is in non-contiguous registers.

Precautions: 1) Do not interrupt during execution.
2) The stack, alpha registers and LAST X are lost during execution.
3) F10 is used and will be cleared at completion.
4) Data must not be greater than E99.
5) S2 will not sort alpha data.

Example 1: Sort the data stored in registers R00 through R15 and list on the 82143A printer.

Do:	See:	Result:
.015	0.015_	Key in bbb.eee
XEQ S2	0.015	After execution
XEQ "PRREGX"	Printer listing	

Example 2: Sort the data in registers R05, R08, R11, R14, R17, R20, R23 (ii = 3)

Do:	See:	Result:
5.02303	5.02303_	Key in bbb.eeeii
XEQ S2	5.02303	After execution

The data has been sorted in the addressed registers with no effect on other data registers.

COMPLETE INSTRUCTIONS FOR S2

S2 will sort random arrays, inverted arrays and sorted arrays in roughly equal time and does not exhibit a large bias toward any particular type. It is somewhat sensitive to the arrangement of data in random arrays, primarily due to provisions that speed execution under certain conditions. (See line by line analysis). Even so, it can be expected to sort most arrays without appreciable differences in execution time.

S2 cannot be interrupted due to use of alpha register P. Interruption will not cause catastrophic results such as MEMORY LOST, but will result in alteration of array data currently held in P. Restarting after interruption will consequently result in the ALPHA DATA message.

As previously stated, sorted arrays are in ascending order. Occasionally, it may be desirable to reverse this order for processing or viewing. Two methods are given in the following examples.

Example 3: The following program may be used to invert an array. It is relatively fast, requiring approximately 42 seconds to invert an array of 300 registers.

If it doesn't look shorter it's because all of the additions required to make S2 do its reassigned task are included. Actually, lines 20, 24-31, 40 and 41 are the rough equivalent of XE. It is fortunate that the necessary return stack information can be left in X when the jump to S2 is made at the STO b instruction.

To run, S22 requires bbb.eee in Y, the number of data to be found before stopping in X, and the ROM entry address in R00. bbb must be R03 or greater.

After constructing S22 in RAM and assigning RCL b and STO M to keys, do the following to find the 3 winners of the pistol match.

Load R03 through R202 with data of your choice and change the sign, either while loading or by the use of CSD after loading.

Do:	See:	Result:
CLA		Find S2 point of entry
GTO S2		address (line 158)
GTO.158		and ASTO in R00.
RCL b		
STO M		
ASTO 00		

3.202 ENTER		Enter bbb.eee	
3		Key the number of data desired	
XEQ "S22"	3.202	After execution	
3.005		Key address of 3 highest data	
XEQ BV		View data. Absolute values in descending order	

APPLICATION PROGRAM FOR: **S2**

```
01♦LBL "S22"          25 RCL b
 02 4                 26 STO [
03 XROM "QR"          27 "├*****"
 04 X≠0?              28 X<> [
 05 SIGN              29 X<> \
 06 +                 30 X<> [
 07 STO 01            31 ARCL 00
 08 X<>Y              32 RCL 02
 09 STO 02            33 STO \
                      34 X<> L
10♦LBL 00             35  E99
 11 INT               36 STO ↑
 12 LASTX             37 ENTER↑
 13 INT               38 ENTER↑
 14 4                 39 R↑
 15 +                 40 X<> [
 16  E3               41 XROM "Sb"
 17 /
 18 +                 42♦LBL 01
 19 INT               43 4
 20 XEQ 14            44 +
 21 DSE 01            45 STO 02
 22 GTO 01            46 GTO 00
 23 RTN               47 2
                      48 END
24♦LBL 14
```

A few final notes: This routine will, of course, find the smallest members of the array if the data signs are not changed. In order to save the bytes required to calculate the M pointer address if an ii increment were involved, only bbb.eee addresses are allowed. To eliminate the use of another data register, this routine does not save bbb.eee. If more than four numbers are specified, the routine will end with the bbb.eee address of the last pass. Since four numbers are selected in each pass through S2, the number of data selected will be the minimum multiple of 4 required to find the number of data specified in X. Note that R00 need be loaded only once if left undisturbed, but don't forget to re-initialize R00 if the ROM port is changed.

```
01♦LBL "IBX"     12 ST+ [       23 +
 02 ENTER↑       13 RDN         24 RCL Y
 03 FRC          14 RCL Y       25♦LBL 00
 04  E3          15 INT         26 X<> IND Y
 05 *            16 +           27 X<> IND Z
 06 INT          17 2           28 X<> IND Y
 07 STO [        18 /           29 ISG Z
 08 LASTX        19 INT         30 DSE Y
 09 FRC          20  E3         31 GTO 00
 10  E3          21 /           32 RTN
 11 /            22 RCL [       33 END
```

After keying in the routine, load registers R01 through R25 with data and then do:

Do:	See:	Result:
1.025	1.025_	Key bbb.eee
XEQ S2	1.025	After execution
XEQ "IBX"	1.025	After execution
XEQ BV		View registers — Array inverted

Example 4: An array can be inverted by changing the sign of all data prior to sorting and again after the sort is complete. To illustrate, key in the following routine:

```
01♦LBL "CSD"      07 X<> IND Y
 02 ENTER↑        08 ISG Y
 03 INT           09 GTO 00
                  10 LASTX
 04♦LBL 00        11 RTN
05 X<> IND Y      12 END
 06 CHS
```

Load data registers R01 through R10 with positive data using BI or other methods then do the following.

Do:	See:	Result:
1.010	1.010_	Key bbb.eee
XEQ "CSD"	1.010	After execution
XEQ S2	1.010	After execution
XEQ "CSD"	1.010	After execution
XEQ BV		View inverted array

Array data is now in descending order in R01 through R10. Positive data was used in the example for simplicity but any data may be used regardless of original sign.

This method is slower than Example 1 but uses less RAM and has a more important application shown in the supportive program for S2.

FURTHER DISCUSSION OF S2

Sort routines are sometimes used on data where only a few of the members are of interest. For example, finding the three highest scores in a pistol match with a field of 200. Handicapped scores are computed to 3 decimal places and sorting by hand is extremely tedious. Here, the 41C is worth its weight in gold! Assign each shooter an ID number, key in the scores as integer numbers with the shooters ID number added as a decimal, sort and give out the prizes.

S3 would take 6.5 minutes to sort the entire field, but we are interested only in three winners. The time to determine them can be cut to 41 seconds if there is some spare RAM available and the following problems are solved:

1) S2 selects the lowest data first and would have to sort the entire array to find the highest data.
2) S2 runs in a continuous loop until finished and cannot be interrupted unless we are sure it already has those 3 numbers.
3) The ROM routine XE could be of some help but on examination it is evident that XE uses and clears the alpha registers. This makes it impossible to enter S2 at some intermediate point since it depends on registers M and N.

The first problem is easily solved by the sign changing trick of Example 4. The shooters ID number will have to be divided by 1000 and added to his score anyway, so the sign can be changed at the same time. Problem #2 can be solved by copying S2 into RAM and changing the register P instructions to R00 instructions. Now, executing in RAM, it can be interrupted periodically to see if register M points above the third register in the array. A VIEW M instruction could be added after LBL04 to eliminate the stops. Alternatively, a loop counter could be added to make S2 jump out after 1 pass and the three numbers (actually 4 or 5) would be there. This is a workable solution that card readers are made to copy and the register P instructions don't have to be changed this way. The cost in RAM, however, will be roughly 140 bytes and the ROM isn't even being used except to copy.

The solution to problem 3 can give a little relief here. It turns out that XE can be copied into RAM and modified by substituting a subroutine call for the CLA at line 129. This allows a subroutine to set alpha registers M and N to needed values. This addition plus the lines necessary for loop counting will cut RAM requirements to 120 bytes but there is a way to cut this even further. **XE** has been designed to keep the stack intact and allow the use of any data register to store the point of entry ROM address. A shorter version will do the job required if these niceties are unneeded.

Routine Listing For: S2

```
144♦LBL "S2"      181 RDN
145 CF 10         182 ISG \
146 STO [         183 GTO 00
147 INT           184 GTO 02
148  E99          185♦LBL 01
149 STO ↑         186 X<> T
150 ENTER↑        187 X<Y?
151 ENTER↑        188 X<>Y
152 ENTER↑        189 R↑
153♦LBL 04        190 X<> ↑
154 X<> [         191 X<> T
155 STO \         192 ISG \
156 X<> [         193 GTO 00
157♦LBL 00        194♦LBL 02
158 X<> IND \     195 R↑
159 X<Y?          196 X<> IND [
160 GTO 01        197 X<> ↑
161 ISG \         198 ISG [
162 GTO 00        199 X<> IND [
163 SF 10         200 R↑
164 GTO 02        201 ISG [
165♦LBL 01        202 X<> IND [
166 R↑            203 R↑
167 X<Y?          204 ISG [
168 GTO 01        205 X<> IND [
169 X<> ↑         206 FS?C 10
170 R↑            207 GTO 03
171 R↑            208 R↑
172 ISG \         209 ISG [
173 GTO 00        210 X<> IND [
174 GTO 02        211♦LBL 03
175♦LBL 01        212 ISG [
176 X<> ↑         213 GTO 04
177 X<Y?          214 LASTX
178 GTO 01        215 RTN
179 RDN           216 END
180 X<> ↑
```

LINE BY LINE ANALYSIS OF S2

In this analysis, the following conventions apply:

1) The term "stack" includes registers X,Y,Z,T and P
2) a,b,c and d represent the four numbers selected as being the smallest found at any intermediate point in a pass through the array. a through d are the smallest through the largest numbers respectively.
3) xn represents the data from register Rnnn. Rnnn is pointed to by register M and N for indirect control.
4) The numbers a,b,c,d and xn are arranged in the stack as:
   - P: b
   - T: a
   - Z: c
   - Y: d
   - X: xn
5) a′, b′, c′ and d′ represent the four numbers selected as being the smallest members of the array during a single pass through the array.

Line 145:	CF10 for useage by routine.
Line 146:	Store bbb.eeeii in M for storage pointer.
Line 147:	Save bbb.eeeii in LAST X.
Lines 148-152:	Fill stack with "dummy" data for initial comparisons until the stack is filled with data from the array.

TECHNICAL DETAILS

XROM: 20,48 — **S2** — SIZE: AS REQUIRED

Stack Usage:
- 0 T: USED
- 1 Z: USED
- 2 Y: USED
- 3 X: USED
- 4 L: USED

Alpha Register Usage:
- 5 M: USED
- 6 N: USED
- 7 O: NOT USED
- 8 P: USED

Other Status Registers:
- 9 Q:
- 10 ⊢:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

NONE USED BY ROUTINE

ΣREG: NOT USED

Data Registers:
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

NONE USED BY ROUTINE

Flag Usage:
04:
05:
06:
07:
08:
09:
10: USED
25:

Display Mode: ANY
FIX 3 or FIX 5 is Recommended.

Angular Mode: N/A

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine:
LBL 00
LBL 01
LBL 02
LBL 03
LBL 04

Execution Time:
$T = 0.273N^{1.438}$ seconds. See curve in text.

Peripherals Required:

Interruptible?	NO
Execute Anytime?	NO
Program File:	S1
Bytes In RAM:	124
Registers To Copy:	47

Other Comments:

Lines 153-156: Initialize recall pointer N from storage pointer M. At the beginning of each pass, M and N will point to the bottom register in the unsorted portion of the array.

Lines 157, 158: Bring xn into X for comparison. The first few executions of this command in each pass will place E99 in the five bottom registers of the unsorted portion of the array.

Lines 159-162: Is xn< d? If so, go to LBL01 to begin insertion in stack. If not, increment recall pointer N and return to beginning of loop to replace xn with next number from array.

Lines 163, 164: If N> eee, set F10 and go to storage instruction sequence.

Lines 165-193: If xn< d, insert in stack in proper location and place d in X. When insertion is completed, increment recall pointer N and return to beginning of loop for next xn. If N> eee, jump to storage sequence but do not set F10.

Lines 194-205: Store a′, b′, c′ and d′ at bottom of unsorted array and simultaneously fill stack with E99. If M> eee at any ISG, skip all subsequent X<> commands to end of routine.

Lines 206-210: If F10 is set, the last number (xn) compared in this pass was greater than d and only a′, b′, c′ and d′ may be stored. If F10 is clear, xn was< d and d is therefore one of the five smallest values found in the pass. Store a′, b′, c′, d′ and xn.

Lines 211-213: Increment storage pointer M. If M> eee skip to end, otherwise go to LBL 04 for another pass.

Lines 214, 215: Return bbb.eeeii to X and RTN.

REFERENCES FOR S2 None known.

CONTRIBUTORS HISTORY FOR S2

S2 is a previously unpublished routine written by Ray Evans (4928) in the search for a fast sort algorithm for the 41C.

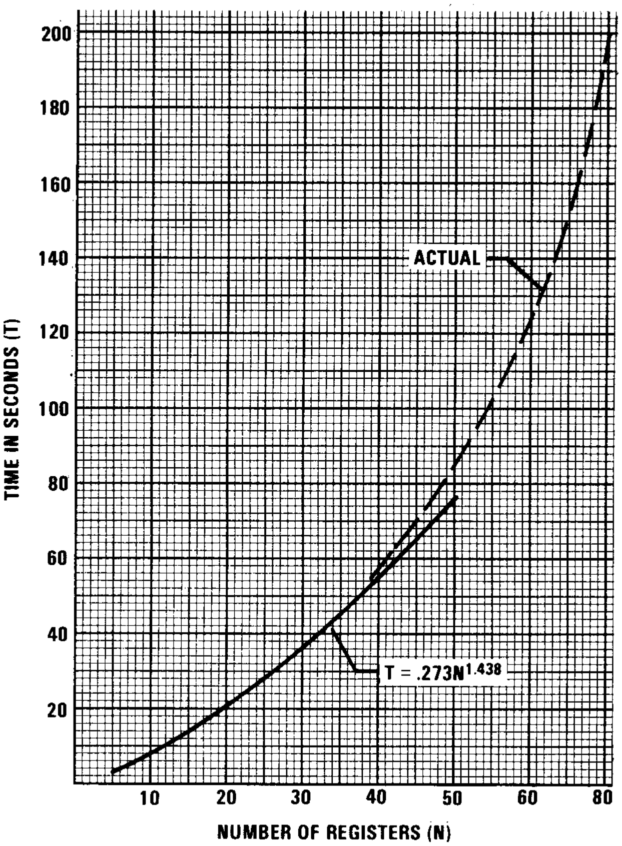
The forerunner of S2 was conceived on the HP-67 but that machine's limited storage capacity made the routine more of a curiosity than a working tool. After joining PPC in 1980 and remembering that the 67 routine would sort small arrays in comparable times to some of the 41C sorts published in PPCJ, the routine was rewritten for the 41C. The original version selected 3 numbers with each pass and other versions followed that saved 4, 5 and 6 numbers per pass. Using the Cheesman Array (PPCJ V6N8P30) as a gauge, the "Save 4" version turned in a time of 2 minutes and 50 seconds, a performance which sparked further effort on code optimization. Many hours and iterations later, the final version evolved. Very slight gains in execution times were still possible but required too many bytes to implement.

Although the versions that saved 5 and 6 numbers per pass were somewhat faster on larger arrays, ("Save 6" Cheeseman array time was 2 minutes) they required too much memory, with the "Save 6" showing definite tendencies toward the point of diminishing returns. When S3 was written, they lost most of their advantage in speed and were dropped.

S3 obviously owes much of its speed to S2 and most 41C array partitioning algorithms would probably benefit from the use of S2 for final sorting. It is estimated that S2 is the culmination of roughly 400 hours spent in development, refinements and testing since original conception.

The author is a mechanical design engineer for E-Systems in Garland, Texas. Until June 1981 his only exposure to programmable machines was through the use of the HP-67 and HP-41C. He is now struggling to learn ATARI BASIC.

S2 TIMING CURVE
Random, Sorted and Inverted Arrays



NOTES:

1) ALL DATA TO GENERATE CURVE WAS TAKEN ON A MACHINE WITH A TIMING CONSTANT OF 1700 ±2. (SEE PPCJ V8N2P32) TO CORRECT FOR A MACHINE WITH A TIMING CONSTANT x, MULTIPLY TIMES GIVEN BY 1700/x.

2) SORTED AND INVERTED ARRAYS WILL RUN WITHIN 3 SECONDS OF TIMES SHOWN FOR RANDOM ARRAYS UP TO 30 REGISTERS. ABOVE N=30, THE TIMES ARE NEARLY IDENTICAL FOR ALL ARRAYS TESTED.

3) RANDOM CURVE BASED ON 27 DATA POINTS USING THREE TEST ARRAYS. THE RANDOM NUMBER GENERATOR WAS OF THE TYPE N_r = FRAC (997∗SEED) WHERE THE SEED IS A SEVEN DIGIT DECIMAL NUMBER AND THE LAST DIGIT IS NOT A MULTIPLE OF 2 OR 5.

4) CURVE WAS FITTED ONLY THROUGH ARRAYS OF 50 REGISTERS. ACTUALS ARE BASED ON ONLY 3 TIMING RUNS OF 50, 64 AND 80 REGISTERS.

FINAL REMARKS FOR S2

In the course of writing documentation for S2 it became obvious that careful consideration should be given the following points during any future ROM effort.

1) It may be a good idea to equip programs using repeating loops with an early exit flag for external control if any forseeable benefits could be derived.

2) The use of the status registers to free data registers from housekeeping chores is good practice but is incompatible with routines such as XE which must use the same registers for decoding, string chopping, etc. Any routine that depends on alpha registers for housekeeping is rendered useless when entered at an intermediate point with XE. It appears to this writer that a great deal of flexibility could be incorporated in the next XE if the CLA command were replaced with an XEQ IND N. In this manner, all sorts of wonderful things could be done with the stack and alpha registers prior to the jump at RTN.

(See Supportive Program for S2 which gives more detail of the difficulties encountered when trying to access a ROM routine to do a job only slightly different from its intended use.)

S3 - LARGE ARRAY SORT (>32)

S3 is used in conjunction with S2 to provide faster execution when sorting large arrays of data in contiguous registers. Its sole purpose is to partition large arrays into blocks of 32 registers (or less) for sorting by S2.

Precautions:
1) Do not interrupt during execution.
2) F10 is used and is clear on completion.
3) The data to be sorted must be in contiguous registers.
4) The array must not contain more than 32 identical data.
5) The data must not be greater than E99.
6) The stack, LAST X, alpha registers, R01 and R02 are used and will be lost.
7) S3 will not sort alpha data.

The array to be sorted is addressed by placing a number of the form bbb.eee in stack register X, where bbb is the beginning data storage register and eee is the ending data storage register. bbb must be R03 or greater. When sorting is complete, bbb.eee is returned to X and the data is sorted in ascending order in registers bbb through eee.

Example 1: Sort the random data in registers R03 through R102.

Do:	See:	Result:
3.102	3.102_	Key in bbb.eee
XEQ S3	3.102	After execution

After approximately 3 minutes the data is sorted in the addressed registers.

Example 2: Sort the random data in registers R010 through R209.

Do:	See:	Result:
10.209	10.209_	Key in bbb.eee
XEQ S3	10.209	After execution

After approximately 6.5 minutes the data is sorted. XEQ "PRREGX" to list the data on the 82143A printer if desired.

APPLICATION PROGRAM 1 FOR S3

Those who have tried alpha sorts on the 41C recognize the agonizing slowness made necessary by the machine's refusal to compare alpha data and its property of destroying NNN's when recalled from user data registers. So far, there are two basic ways to accomplish alpha comparisons. The first is to recall the alpha string and convert it to an NNN for comparison in the status registers where, fortunately, it is not normalized. The second method involves converting the string to a numeric equivalent capable of being stored and recalled at will and then reconverting the number to the original alpha string when comparisons are complete.

Using the first method, the conversion must be repeated many times on the same string during the course of sorting a large number of alpha data and is responsible for the long times involved. The second method may or may not be better due to conversion and reconversion times for six character alpha strings. Much depends on the number of data to be compared. The following routine using the second method and will do a reasonable job on large arrays but contains severe compromises to reduce execution time:

1) It does not use the ASCII decimal equivalent for a character.
2) It is limited to the upper case alphabet, a-e, the alpha numerals 0-9, e space, $, / and % characters. (Hex codes 20-29, 30-39, 40-4F, 50-5F and 60-6F.)

The routine requires a total of approximately 5 seconds per register to convert and re-convert the alpha data. Using S3, it is capable of sorting an average array of 100 character strings in approximately 11 minutes. This won't set the calculator on fire but is presented to fill a gap until HP or assembly language come along.

APPLICATION PROGRAM FOR: **S3**

```
01♦LBL "S3A"
02 STO 00
03 STO Z

04♦LBL 10
05 CLA
06 ARCL IND 00
07 "├     "
08 ASTO X
09 CLA
10 ARCL X
11 "├♦♦♦♦"
12 CLX
13 X<> [
14 ASTO Y
15 X<> \
16 ASTO X
17 XEQ 05
18  E-4
19 *
20 X<>Y
21 XEQ 05
22  E2
23 *
24 FRC
25 ST+ Y
26 X<> L
27 INT
28  E
29 +
30 10↑X
31 *
32 STO IND 00
33 ISG 00
34 GTO 10
35 RDN
36 GTO 15

37♦LBL 05
38 X<> d
39 CF 03
40 FS? 11
41 SF 10
42 FS? 19
43 SF 18
44 FS? 27
45 SF 26
46 X<> d
47 RTN

48♦LBL 15
49 XROM "S3"
50 STO 00

51♦LBL 00
52 CLA
53 X<> IND 00
54 STO [
55 ASTO [
56 "├←"
57 X<> [
58 FRC
59 LASTX
60 INT
61  E4
62 +
63 X<> [
64 "├♦"
65 STO \
66 ASHF
67 "├♦♦"
68 X<>Y
69  E
70 +
71 STO [
72 RCL \
73 "├♦"
74 STO \
75 "├♦♦♦♦"
76 XEQ 05
77 XEQ 05
78 XEQ 05
79 XEQ 05
80 XEQ 05
81 XEQ 05
82 X<> IND 00
83 ASTO IND 00
84 ISG 00
85 GTO 00
86 RTN

87♦LBL 05
88 X<> \
89 X<> d
90 FS?C 00
91 GTO 02
92 FC? 01
93 GTO 03
94 FC? 03
95 GTO 03
96 FS?C 02
97 GTO 03
98 CF 03
99 GTO 04

100♦LBL 02
101 SF 01
102 SF 03

103♦LBL 04
104 SF 04
105 FS? 06
106 SF 05
107 FC?C 06
108 SF 06

109♦LBL 03
110 X<> d
111 X<> \
112 "├♦"
113 END
```

After keying in the routine (line 07 is 5 spaces, line 56 is Hex F2, 7F, 03) and filling registers R03 through R52 with alpha strings, do the following:

Do:	See:	Result:
3.052	3.052_	Key in bbb.eee
XEQ "S3A"	3.052	After execution

After 5 to 5.5 minutes, the alpha data will be sorted in ascending order in R03 through R52.

A few additional notes on the routine:

1) bbb must be R03 or greater when using S3 as listed.
2) If S2 is used instead of S3, bbb may be R01 or greater.
3) If S2 is used, an address including the increment ii may be used.
4) When strings of less than 6 characters are entered, spaces are appended to fill out the string. At least one space is required since the first character is placed as the exponent in the decimal conversion and requires a mantissa.

Routine Listing For: S3

```
 34♦LBL "S3"          90 +
 35 STO ]             91♦LBL 10
 36 SF 10             92 ENTER↑
 37♦LBL 09            93 FRC
 38 STO 02            94  E3
 39♦LBL 05            95 *
 40 STO 01            96 X<>Y
 41 FRC               97 -
 42  E3               98 31
 43 *                 99 X>Y?
 44 STO \            100 GTO 01
 45 RCL 01           101 LASTX
 46 INT              102 FS?C 10
 47 -                103 GTO 09
 48  E               104 GTO 05
 49 +                105♦LBL 01
 50 STO [            106 LASTX
 51 RCL 01           107 XROM "S2"
 52  E-4             108 INT
 53 +                109 RCL ]
 54 .                110 INT
 55 RCL X            111 X=Y?
 56♦LBL 14           112 GTO 08
 57 RCL IND Z        113 RDN
 58 +                114 RCL 01
 59 ISG Y            115 INT
 60 ""               116 X<>Y
 61 ISG Z            117 X>Y?
 62 GTO 14           118 GTO 07
 63 X<>Y             119 RCL 02
 64 /                120 INT
 65 RCL \            121 X<>Y
 66 X<>Y             122 X>Y?
 67 RCL 01           123 GTO 07
 68 STO T            124  E
 69♦LBL 13           125 -
 70 X<> IND \        126  E3
 71♦LBL 11           127 /
 72 X<Y?             128 RCL ]
 73 GTO 12           129 INT
 74 DSE \            130 +
 75 ""               131 SF 10
 76 DSE [            132 GTO 09
 77 GTO 13           133♦LBL 07
 78 GTO 06           134  E
 79♦LBL 12           135 -
 80 X<> IND T        136  E3
 81 ISG T            137 /
 82 ""               138 +
 83 DSE [            139 GTO 10
 84 GTO 11           140♦LBL 08
 85♦LBL 06           141 LASTX
 86 X<> IND Z        142 BEEP
 87 FRC              143 RTN
 88 R↑
 89 INT
```

S3's sole function is to partition large arrays of data in contiguous registers into blocks of 32 or less registers for processing by S2. This task is accomplished by summing every tenth register in the array (or sub-array) to find an approximate average. All numbers equal to or greater than the average are then relocated to the upper registers and all numbers less than the average are moved to the lower registers. This will, ideally, split the array in half. This process is repeated, if necessary, until a block of 32 or less registers has been produced at the top of the array and S2 is called. The routine then repeats this process on descending blocks of registers pointed to by R01 and R02 until sorting is complete.

Although S3 was written to obtain faster execution on large arrays (as indicated by its title), it will sort contiguous arrays of any size thus eliminating the necessity of calling different routines based on array size. S2, however, will generally sort the smaller arrays in equal or less time (depending on the data), with no restrictions on bbb and the ability to sort non-contiguous arrays.

As mentioned previously, S3 will not sort arrays containing more than 32 identical data. As a result of the attempt to optimize execution time, the S3 algorithm requires blocks of 32 registers or less to call S2 and will go into an endless "do-loop" when asked to sub-divide data that cannot be partitioned further.

The precaution against interruption is due only to the use of register P in the S2 subroutine. The user may find after some experimentation that he can "read" the goose to determine when S3 may be interrupted without loss of data, i.e., when S2 is not executing. If loss of data is unimportant, S3 may be interrupted without catastrophic results such as "MEMORY LOST" but may display "Alpha Data" if restarted.

The design of S3 contains byte and/or register saving compromises which increase execution time to some degree. R01 and R02 contain only 2 partitioning addresses but could easily have recorded 4 addresses at the expense of the bytes required to assemble and read them. This would have prevented some unnecessary backtracking on very large arrays. Alternatively, a list of "stacks to sort" could have been generated but these lists are variable in size and this disadvantage was deemed to outweigh whatever increase in speed might accrue. Steps could have been incorporated to eliminate the inability of S3 to sort arrays containing more than 32 identical data but again, the bytes necessary were not thought practical for such rare occurances.

LINE BY LINE ANALYSIS OF S3

Line 35:	Save bbb.eee in register "O".
Line 36:	Initialize F10 to save first partition address in R02 until upper "half" of array is sorted.
Line 37-48:	Save partition addresses in R01 and R02
Line 41-50:	Calculate number of registers in array. Place eee in N and number of registers in M.
Line 51-53:	Add ii increment to bbb.eee to sample every 10th register in array.
Line 54-62:	Sum every 10th register.
Line 63-68:	Calculate approximate average of data in array and set up stack as: T: bbb.eee Z: eee Y: average X: bbb.eee
Line 69-84:	Move all numbers less than the average to the bottom of the array. Move all numbers greater than or equal to the average to the top of the array. After the skip at DSE M, T will contain the address of the lowest register containing data equal to or greater than the average.

TECHNICAL DETAILS

XROM: 20,47	S3	SIZE: AS REQUIRED

Stack Usage:	Flag Usage:
0 T: USED	04:
1 Z: USED	05:
2 Y: USED	06:
3 X: USED	07:
4 L: USED	08:
Alpha Register Usage:	09:
5 M: USED	10: USED
6 N: USED	
7 O: USED	
8 P: USED	25:

Other Status Registers:	Display Mode: ANY
9 Q:	FIX 3 Recommended.
10 ⊢:	
11 a: NONE USED BY	Angular Mode: N/A
12 b: ROUTINE	
13 c:	
14 d:	Unused Subroutine Levels:
15 e:	4

ΣREG: NOT USED	Global Labels Called:	
Data Registers:	Direct	Secondary
R00: NOT USED	S2	NONE
R01 & R02: USED		
R06:		
R07: Other Registers are used for array storage.		
R08:		
R09:		
R10:		
R11:	Local Labels In This Routine:	
R12:	LBL 01	
	LBLS 05 - 14	

Execution Time:
T = $0.517N^{1.26}$ seconds. See curve in text.

Peripherals Required: NONE

		Other Comments:
Interruptible?	YES*	*S3 is interruptible, it calls S2, which is not. See text.
Execute Anytime?	NO	
Program File:	S1	
Bytes In RAM:	159	
Registers To Copy:	47	

Line 86-97: Calculate the number of registers between the address in T and eee.

Line 98-100: If number of registers is equal to or less than 32, go to LBL 01 for processing by S2.

Line 101-104: If the number of registers is greater than 32, return to LBL 05 or 09 to partition the upper portion of the array again. If this is the original partition, save the T address in R02. Any subsequent partition addresses will be stored in R01.

Line 105-107: Call S2 for sorting.

Line 108-112: Compare lowest register in block just sorted to bbb. If equal, exit to LBL 08.

Line 113-118: Compare lowest register in block just sorted to address saved in R01. If greater, GTO 07 to calculate block address.

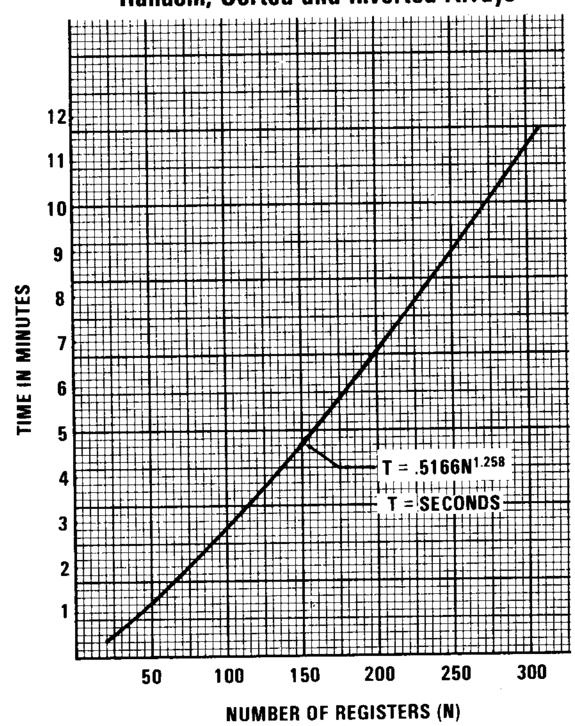
Line 119-123: Compare lowest register in block just sorted to address saved in R02. If greater GTO 07 to calculate block address.

Line 124-132: Upper "half" of array has been sorted. Calculate address of block between bbb and block just sorted. SF10 to save first partitioning of lower "half" of array in R02 and GTO 09 to start partitioning. Note that routine will partition lower half of array even if it contains 32 or fewer registers. This is a byte saving compromise due to structure between lines 37 and 104.

Line 133-139: Block address calculations used in lines 113-123. Return to LBL 10 to check block size.

Line 140-143: Return bbb.eee to X, BEEP and RTN.

S3 TIMING CURVE
Random, Sorted and Inverted Arrays



NOTES:

1) ALL DATA TO GENERATE CURVE WAS TAKEN ON A MACHINE WITH A TIMING CONSTANT OF 1700 ±2. (SEE PPCJ V8N2P32) TO CORRECT FOR A MACHINE WITH A TIMING CONSTANT x, MULTIPLY TIMES GIVEN BY 1700/x.

2) RANDOM CURVE BASED ON 23 DATA POINTS USING THREE TEST ARRAYS. THE RANDOM NUMBER GENERATOR WAS OF THE TYPE N_r = FRAC (997 * SEED) WHERE THE SEED IS A SEVEN DIGIT DECIMAL NUMBER AND THE LAST DIGIT IS NOT A MULTIPLE OF 2 OR 5.

REFERENCES FOR S3

None known.

CONTRIBUTORS HISTORY FOR S3

S3 is a previously unpublished routine written by Ray Evans (4928) in support of the S2 program.

S3 was written in the waning weeks before ROM finalization and was transcribed to Richard Nelson via telephone when rates are cheap but the coffee isn't. Apparently the written listings and mag cards never arrived in California.

It is estimated that S3 is the result of approximately 100 hours spent in writing and refinement. The earliest version split an array into four parts regardless of size but was found lacking in speed when array sizes were greater than ≈ 128. The greatest portion of time was spent in reducing byte count and optimizing the code for faster execution.

FINAL REMARKS FOR S3

The remarks contained in the "Blue Sky Observations" for S2 pertain to S3 as well. So far, S3 has been found totally impossible to enter at an intermediate point by programs such as S22 to obtain any practical benefit. The only exit provision contained in S3 is a comparison of bbb with the beginning register number of the last block sorted by S2. This address varies with each array until sorting is complete and cannot be used to provide an exit at the completion of intermediate block sorting. Had an early exit flag been incorporated, S3 would quickly have found the highest numbers in an array without resorting to the sign changing necessary in S22.

In the course of ROM timing runs and program documentation, data was taken which indicated that a slight optimization error was incorporated in S3. This data points to a block size of 35 instead of 32 for processing by S2. The effect of this miscalculation is quite small and would not likely be noticeable until timing for a large number of runs on random data was accumulated and averaged.

NOTES

S? - SIZE FINDER

The SIZE Finder function S? is the classic example of a mainframe function that HP forgot. For instance, you try to RCL a register and it comes up NONEXISTENT; you try a lower register and it's NONEXISTENT too. What is the SIZE, anyway? Isn't there any easy way to find out? Well, if you have a printer you can XEQ "PRFLAGS". Otherwise you have to think of something else.

With the PPC ROM in place you just XEQ S? to find out the current SIZE.

Example 1: XEQ "SIZE" 10 then XEQ S? . The result is 10. You can try this example for any SIZE and S? will give the correct result.

COMPLETE INSTRUCTIONS FOR S?

XEQ S? to get a decimal number in x indicating the number of data registers allocated. The original X and Y are preserved in Y and Z. Z, T, and L are lost.

MORE EXAMPLES OF S?

Example 2: Suppose you want to set the ΣREG block to the highest available data registers. This can be done using the following sequence:

```
XROM S?
6
-
ΣREG IND X
```

Example 3: When you're loading in programs and all of a sudden you get PACKING/TRY AGAIN, XEQ S? to check the current SIZE; you'll see how much leeway you have for decreasing the SIZE to allocate more registers to program memory.

LINE BY LINE ANALYSIS OF S?

S?, like all synthetic SIZE finders, extracts the curtain absolute address from status register c. Next it calculates x = (-curtain) mod 64. This is the SIZE modulo 64. The actual SIZE will be x, x + 64, x + 128, x + 192, or x + 256. The correct SIZE is determined by checking whether data register x + n * 64 is NONEXISTENT.

32 LBL S?	
33 XEQ C?	obtain absolute address of curtain
34 CHS	change sign of curtain address
35 LBL 13	label used by ML
36 64	represents one memory module (single density)
37 MOD	place starting size in x, store 64 in L
38 SF 25	set flag 25 for detection of NONEXISTENT
39 LBL 02	label
40 RCL IND X	check to see if NONEXISTENT
41 FC? 25	
42 RTN	if yes, return to user
43 X<>L	if not, add 64 and try again
44 +	
45 GTO 02	

CONTRIBUTORS HISTORY FOR S?

PPC members started writing SIZE finders almost as soon as the 41C came out. Brute force linear search techniques gave way to the binary search algorithm used by Ron Knapp's (618) elegant routine (see *PPC CALCULATOR JOURNAL)*, V7N2P38d). Synthetic versions provided greatly increased speed at the expense of a higher byte count. Roger Hill's (4940) version of V7N5P57d is the ultimate in speed. The ROM version, written by Keith Jarett (4360), sacrifices some speed for byte savings. The byte savings are realized by combining S?, C?, and Σ?.

Routine Listing For: S?	
32♦LBL "S?" 33 XROM "C?" 34 CHS 35♦LBL 13 36 64 37 MOD 38 SF 25	39♦LBL 02 40 RCL IND X 41 FC? 25 42 RTN 43 X<> L 44 + 45 GTO 02

FURTHER ASSISTANCE ON S?

Call Keith Jarett (4360) at (213) 374-2583.
Call Roger Hill (4940) at 656-8825.

NOTES

NOTES

TECHNICAL DETAILS

XROM: 10,15	S?	SIZE: 000

Stack Usage:
- 0 T: Y
- 1 Z: Y
- 2 Y: X
- 3 X: SIZE
- 4 L: 64

Flag Usage: MANY USED BUT ALL RESTORED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10:
- 25:

Alpha Register Usage:
- 5 M:
- 6 N: ALL USED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 5

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R06:
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Global Labels Called:

Direct	Secondary
C?	NONE

Local Labels In This Routine:

02
13

Execution Time: 1.4 - 2.0 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	YES
Program File:	ML
Bytes In RAM:	26
Registers To Copy:	64

Other Comments:

SD - STORE DISPLAY MODE

SD saves flags 16-55 in a register defined by X, so that the user may then change the contents of register d (particularly display format and trig mode) during a program yet, have the capability of restoring the original format at the end of the program by by calling RD. Although SD will be used primarily by programs, it can be executed from the keyboard if necessary.

Example 1: SD is used on line 129 in the First Derivative routine FD to store the user's display status prior to changing it. RD is used on line 168 to restore the user's display status.

Example 2: XEQ RF to set default flag status. Then key in 0, XEQ SD. Now set flag 04, set ENG 7, GRAD mode, and switch to USER mode. Key in 0, XEQ RD to restore the original (default) flags, except for flag 04 which remains set.

COMPLETE INSTRUCTIONS FOR SD

1. Insert into X a register number.
2. XEQ SD
3. After execution of the program, the register that was chosen will contain an alpha string consisting of a star and the final five bytes of register d.
4. The same register number, followed by XEQ RD, will restore the previous status of flags 16-55 (just prior to XEQ SD).
5. SD saves Y, Z, and T in X, Y, and Z. X is placed in L.

MORE EXAMPLES OF SD

Example 3: As a more exotic application, if you find yourself switching between several different display formats quite often, it may be useful to use SD to create a block of registers that contain different flag settings, e.g.,

R20 = FIX 2, DEG MODE, SF 28, SF 29
R21 = ENG 3, RAD MODE, SF 28, CF 29

etc. so these different display settings could be called up (by RD) as needed. These registers would be created by actually setting the display as wanted and then using SD to store into the desired location.

LINE BY LINE ANALYSIS OF SD

45.	LBL SD	
46.	SIGN	store X in L
47.	RDN	restore stack
48.	RCL d	place register d in X
49.	STO M	place register d in alpha
50.	"├--"	append 2 nulls to shift 1st 2 bytes of d into N
51.	X<>M	X = flags 16-55 of d + 2 null bytes
52.	GTO 14	jump to Label 14
62.	LBL 14	label 14
63.	"*"	place star in alpha
64.	X<>M	place star in X put flags 16-55 + 2 nulls in M
65.	STO N	place star in N
66.	ASTO IND L	cause star + flags 16-55 to be ASTO'd in designated register
67.	RDN	restore stack
68.	RTN	

CONTRIBUTORS HISTORY FOR SD

The first display mode save/recall routines (see *PPC CALCULATOR JOURNAL*, V7N5P8) were apparently written by Leigh Borkman (5218). SD was written by Keith Jarett (4360) as an addition to SK (because of the shared code).

TECHNICAL DETAILS

XROM: 20,03	SD	SIZE: 001

Stack Usage:
0 T: USED
1 Z: T
2 Y: Z
3 X: Y
4 L: X

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Alpha Register Usage:
5 M:
6 N: ALPHA STORED IN
7 O: R_X
8 P:

Other Status Registers:
9 Q:
10 ├: NONE USED
11 a:
12 b:
13 c:
14 d:
15 e:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

ΣREG: UNCHANGED

Data Registers: ONE REGISTER SPECIFIED BY USER IS USED FOR STORAGE

Global Labels Called:

Direct	Secondary
PART OF SK	NONE

Local Labels In This Routine: NONE

Execution Time: .4 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	SR
Bytes In RAM:	20
Registers To Copy:	40

Other Comments:

Routine Listing For: SD

```
45♦LBL "SD"        62♦LBL 14
46 SIGN            63 "*"
47 RDN             64 X<> [
48 RCL d           65 STO \
49 STO [           66 ASTO IND L
50 "├♦♦"           67 RDN
51 X<> [           68 RTN
52 GTO 14
```

FURTHER ASSISTANCE ON SD

Call Keith Jarett (4360) at (213) 374-2583.
Call Keith Kendall (5425) at (801) 967-8080.

APPENDIX J CONTINUED FROM PAGE 387.

PPC ROM FILE (GROUP) BRANCHING TABLE

VM GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
CD			10001		VM		
CU			003	MT			
CX			006				V
DS			009				V
EP			21013				R
EX			014		EX		
MT			028				R
TN			029		MT		
VM			041				R
VS			042		DS		
----			058				R
00			059		VS		
01			068				C R
04			078				R
13-2			079		EP		
14-2			081			//	RAM
		14	084				C
			085	PD			
		13	091				
	14		092				
			093	C?			
	13		094				
			095	E?			
			22102	OM			
			105			04	C
			112	GE			
	04		113				
		04	116				C
			117				R
			118		TN		
			120	DC			
			125	XE			
			127				R
			128		CX		
			129	C?			
			131		CU		
	00		141				
		14	146				C
		13	153				C
	01		154				
		13	160				C
		01	164				C
	13		165				
		00	167				C
	14		168				
			177				R
			178		CD		
			23205				E

Groups are in ROM order except for SR & NS. This was done to conserve space. These listings contain mostly those lines considered necessary to keep track of branch or return stack levels.

VK GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
AL			23001		VK		
NC			006				R
SU			010			07	
VK			015			07	
----	01		019				
01	02		025				
02		05	028				C
03		05	032				C
04	03		034				
05		02	036				C
06		01	038				C
07	04		040				
12-2			046				R
13	05		047				
14-4			063				a
			069				a
		04	24085				C
		06	094				C
	06		097				
			100				A
		03	112				
	07		113				
			119				R
			120		AL		
			123			14	
			124			14	
			126			13	C
		12	132				C
			134				C R
			138				R
	12		139				
		12	141				C
	12		144				
			147				R
	13		148				
			152			14	
	14		153				
		14	25164				C
	14		167				
			171				R
			172		NC		
		14	174				
			175		SU		
	14		177				
		14	192				C
			221				R
	14		222				
			227				E

Due to multiple usage of certain local labels it is difficult to follow jump instructions on the regular ROM listing. Because of the column format used here you can pick out jump and local label relationships at a glance.

NH GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
HN			25001		NH		
NH		12	024				C
----	01		26027				
01		14	033				C
02	13		034				
12		14	040				C
13		14	043				C
14-3	14		046				
		01	052				C
	12		053				
			060				R
			061		HN		
	02		064				
		14	071				C
		14	076				C
	14		081				
		14	091				C
		14	096				C
	14		101				
		02	110				C
			27114				E

Column Headings Defined

LBLS=Global labels in alphabetical order & local labels in numerical order.
LL=Local labels in ROM order.
GT=GTO commands in ROM order.
PCLn#=Address from ROM listing modified as follows:
P=Page number 1 thru 5.
C=Column number 1 thru 9 & 0
0 stands for column #10.
Ln#=Line number 001 thru 257
Example-10001 is page 1, column 10, line 001.
XR=XROM commands in ROM order.
GL=Global labels in ROM order.
XQ=XEQ commands in ROM order.
CIS= Breaks down as follows:
C-Conditional test preceeds the command on this line.
I-Command is indirect.
S- As follows:
A-AVIEW R-RTN
E-END S-STOP
P-PROMPT V-VIEW
RAM-Calls RAM routine
a-May alter status reg a.
b-May alter status reg b.

BL GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
BI			27001		BL		
BL	02		007				
FL			012	QR			
IP			017				S
MS		02	020				
PS			021		FL		
T1	00		024				
----			025				S
00			028	QR			
01	01		029				
02		13	032				C
10			043			14	C
13-2		00	046				
14-3	13		047				
			050			14	
		01	052				
	14		053				
			059				S
			060				R
			061		BI		
	10		063				
		10	068				C
			069				R
			070		IP		
			071			14	
			075				R
			076		PS		
			081			14	
			28098				C P
		13	100				C
			102				C R
			103	TN			
			104	TN			
	13		107				
			113			Z	I
			114	GE			
	14		115				
			116	E?			
		14	120				C
			126	OM			
			132				R
	14		133				
			135	S?			
			138	VA			
			139	GE			
			140		T1		
			154				R
			155		MS		
			168				E

IF GROUP

LBLS	LL	GT	PCLn#	XR	GL	XQ	CIS
2D			28001		IF		
Ab			29031				R
CB			032				R
DP			033		CB		
E?			035	PD			
GE			037	PD			
IF			039				R
OM			040		RT		
PA			044	2D			
PD		14	051				
QR			052		PD		
RT			053	2D			
RX			055	QR			
SX	14		058				
----			066				R
13			067		DP		
14-3			069	QR			
			074	QR			
			078	DC			
			079	DC			
			081				R
			082		QR		
			093				R
			094		2D		
			20121				R
			122		SX		
			123			14	
		13	128				
			129		RX		
			130			14	
	13		132				
			136				R
	14		137				
			142		OM		
			143			14	
			151				R
			152		PA		
			154	PD			
			157	DP			
			158		GE		
			163				C R
			166			14	
			181		Ab		
			182				b
	14		31183				
			194				R
			195		E?		
			197	2D			
			206				E

END

SE - SELECTION WITHOUT REPLACEMENT

The selection without replacement routine can be used to select at random an element from any block of consecutive registers. Subsequent items selected from the block will not be repeated. The data block that this routine selects from can have its data arranged in any order. This routine can also be considered as a random shuffler which will scramble the contents of a block of registers. SE calls the random number generator routine RN.

Example 1: Use SE to select at random names from the following list of names.

First store the names in the registers indicated below.

```
R15: Joe
R16: Mary
R17: Bob
R18: John
R19: Bill
R20: Jane
R21: Robert
R22: Dick
R23: George
R24: Tracy
R25: Mark
R26: Ann
R27: Susan
```

See the routine RN. SE calls RN and hence requires a register to hold random decimals. For this example we will use R05 which must first be initialized with a random decimal. Store 0.141592654 in R05.

SE requires the following information about the block to be selected from. In R06 store the number of the first register in the block. In R07 store the number of registers within the block.

For this example store 15 in R06 and store 13 in R07. Once the random number generator has been initialized and the constants have been stored in R06 and R07 the normal input to SE is simply the number of the register which holds the random number generator seeds. For this example we are using R05 so key in 5 and XEQ "SE".

The first name selected is Mark. Each time SE is called the item selected is exchanged with the item at the bottom of the list and the number in R07 is decreased by one. In this manner the selected item cannot be selected again since R07 determines the upper limit of the selection range. In this same manner the block of data becomes rearranged. After selecting Mark the register contents are:

```
R07: 12      R15: Joe
             R16: Mary
             R17: Bob
             R18: John
             R19: Bill
             R20: Jane
             R21: Robert
             R22: Dick
             R23: George
             R24: Tracy
             R25: Susan
             R26: Ann          =new last element
             R27: Mark         =previous selection
```

Now key in 5 and XEQ "SE" to select another name. This time Mary is selected and the register contents change to the following:

```
R07: 11      R15: Joe
             R16: Ann
             R17: Bob
             R18: John
             R19: Bill
             R20: Jane
             R21: Robert
             R22: Dick
             R23: George
             R24: Tracy
             R25: Susan        =new last element
             R26: Mary         =previous selection
             R27: Mark         =already used
```

Note how the most recently selected item is placed at the bottom of the new list and how the item at the previous bottom of the list is exchanged with the current selection.

Now key in 5 and XEQ "SE" to select yet another name. Robert is selected this time and the register contents change to the following:

```
R07: 10      R15: Joe
             R16: Ann
             R17: Bob
             R18: John
             R19: Bill
             R20: Jane
             R21: Susan
             R22: Dick
             R23: George
             R24: Tracy        =new last element
             R25: Robert       =previous selection
             R26: Mary         =already used
             R27: Mark         =already used
```

Now key in 5 and XEQ "SE" to select another name. This time Bill is selected and the register contents change to the following:

```
R07: 9       R15: Joe
             R16: Ann
             R17: Bob
             R18: John
             R19: Tracy
             R20: Jane
             R21: Susan
             R22: Dick
             R23: George       =new last element
             R24: Bill         =previous selection
             R25: Robert       =already used
             R26: Mary         =already used
             R27: Mark         =already used
```

We can continue to select elements as long as the number in R07 is positive. Continuing to key in 5 and XEQ "SE" the following names are selected in order. Joe, George, John, Tracy, Ann, Susan, Bob, Dick, and Jane. After selecting the last name, Jane, the contents of the registers are as follows:

```
R07: 0       R15: Jane
             R16: Dick
             R17: Bob
             R18: Susan
             R19: Ann
             R20: Tracy
```

R21: John
R22: George
R23: Joe
R24: Bill
R25: Robert
R26: Mary
R27: Mark

If this list is compared with the original at the beginning of this example we can see that the effect of applying SE 13 times is to re-arrange the list of names in a random order.

COMPLETE INSTRUCTIONS FOR SE

1) Three registers must be initialized before SE can be called for the first time. The list to be selected from must be in consecutive registers and the number of the first register should be stored in R06. The number of registers in the block should be stored in R07.

2) SE calls the random number generator routine RN and RN requires one data register to hold random decimals. See the RN routine. Let register k be used to hold the random decimals. Before SE is called the first time store a random decimal in the range 0-1 in register k.

3) Once initialized, the normal input to SE is simply the number k which points to the random decimal register. Key in k and XEQ "SE".

4) The output from SE is the register content chosen at random and is left in the X-register. Each time SE is called the counter in R07 is decreased by one. Items selected will not repeat.

5) If many calls are being made to SE then R07 should be tested for a zero value before SE is called. When R07 is zero all the available items will have been selected. The items remain stored in the original block of registers but they will be re-arranged.

6) To begin a whole new selection process only the number in R07 needs to be re-initialized. The stack input/output for SE is as follows:

Input:		Output:	
T:	T	T:	T
Z:	Z	Z:	reg. # selected
Y:	Y	Y:	pointer last reg.
X:	RN pointer	X:	item selected
L:	L	L:	# remaining items

MORE EXAMPLES OF SE

Example 2: Store the following data in R10-R20.

R10: 10	R16: 16
R11: 11	R17: 17
R12: 12	R18: 18
R13: 13	R19: 19
R14: 14	R20: 20
R15: 15	

Apply SE 11 times to select all the numbers from R10-R20.

Key 10.02 ENTER↑ 10 ENTER↑ 1 and XEQ "BI" to store the data.

Store the number of the first register in the block = 10 in R06. Then store the number of registers = 11 in R07. As in the previous examples using RN we will use R05 to hold the random decimals and we will initialize R05 with 0.

Perform the following step 11 times: Key in 5 and XEQ "SE".

The contents of R10-R20 after 11 selections is:

R10: 15	R16: 13
R11: 14	R17: 20
R12: 10	R18: 11
R13: 17	R19: 16
R14: 19	R20: 12
R15: 18	

To view the data key 10.02 and XEQ "BV". Reading the data backwards from R20 down to R10 gives the order of the numbers selected.

Now re-initialize only R07 with 11 and make another 11 selections by keying in 5 and XEQ "SE" 11 times. Now the contents of R10-R20 are:

R10: 18	R16: 20
R11: 13	R17: 10
R12: 15	R18: 12
R13: 17	R19: 16
R14: 11	R20: 19
R15: 14	

APPLICATION PROGRAM 1 FOR SE

This routine shows how SE may be used to shuffle or randomly scramble the elements in a block of data. An obvious application is to shuffle a deck of cards, although SE can be used to deal cards at random without shuffling. The input to SHUFFLE is a block control word of the form bbb.eee which describes a block of consecutive registers. Since SHUFFLE calls SE which in turn calls RN, the random number generator must be initialized with a starting seed. SHUFFLE uses R05 to hold the seeds. First store a random decimal in R05. Then key in bbb.eee and XEQ "SHUFFLE". When the program ends the register contents of the block will be in random order.

```
LBL*SHUFFLE
INT
STO 06
ST - L
LAST X
E3
*
X<>Y
-
1
+
STO 07
LBL 01
5
```

```
XROM SE
RCL 07
X≠0?
GTO 01
RTN
```

Example: Assume registers R11-R62 hold the numbers 1-52 in order.

R11= 1	R24= 14	R37= 27	R50= 40
R12= 2	R25= 15	R38= 28	R51= 41
R13= 3	R26= 16	R39= 29	R52= 42
R14= 4	R27= 17	R40= 30	R53= 43
R15= 5	R28= 18	R41= 31	R54= 44
R16= 6	R29= 19	R42= 32	R55= 45
R17= 7	R30= 20	R43= 33	R56= 46
R18= 8	R31= 21	R44= 34	R57= 47
R19= 9	R32= 22	R45= 35	R58= 48
R20= 10	R33= 23	R46= 36	R59= 49
R21= 11	R34= 24	R47= 37	R60= 50
R22= 12	R35= 25	R48= 38	R61= 51
R23= 13	R36= 26	R49= 39	R62= 52

Use BI to load the registers. Key 11.062 ENTER↑ 1 ENTER↑ XEQ " BI ".

Store .141592654 in R05 as the starting seed. Then key in 11.062 and XEQ "SHUFFLE". After about 63 seconds when the routine ends the contents of the block are scrambled:

R11= 11	R24= 25	R37= 8	R50= 44
R12= 27	R25= 6	R38= 29	R51= 51
R13= 28	R26= 48	R39= 33	R52= 45
R14= 36	R27= 14	R40= 31	R53= 41
R15= 20	R28= 26	R41= 52	R54= 16
R16= 34	R29= 35	R42= 1	R55= 50
R17= 5	R30= 24	R43= 4	R56= 22
R18= 15	R31= 46	R44= 37	R57= 2
R19= 19	R32= 10	R45= 47	R58= 3
R20= 30	R33= 13	R46= 39	R59= 23
R21= 40	R34= 38	R47= 12	R60= 32
R22= 9	R35= 17	R48= 43	R61= 7
R23= 18	R36= 49	R49= 21	R62= 42

Use BV to inspect the results. Key 11.062 and XEQ " BV ".

Routine Listing For: SE

```
18♦LBL "SE"         25 ""
19 XROM "RN"        26 RCL 07
20 RCL 07           27 +
21 *                28 RCL IND X
22 RCL 06           29 X<> IND Z
23 ST+ Y            30 STO IND Y
24 DSE 07           31 RTN
```

LINE BY LINE ANALYSIS OF SE

SE is a short routine which first calls RN. The resulting random number is rescaled to produce a pointer in the range between the elements in the block that remains. A pointer to the last element in the block is used to perform the register exchange at the time the selection is made.

REFERENCES FOR SE

Bill Kolb (265), "PPC Journal," Selection Without Replacement, V5N1P5b

CONTRIBUTORS HISTORY FOR SE

Selection without replacement was brought to light by Bill Kolb (265) in his article in V5N1P5b. John Kennedy (918) wrote the HP-41C version which takes advantage of the 41C's extended addressing modes as well as calls on the RN routine. John also wrote the documentation for SE

FURTHER ASSISTANCE ON SE

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

NOTES

TECHNICAL DETAILS

XROM: 20, 56 | **SE** | **SIZE:** depends on data block

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:

The data registers depends on the initial block used and the register used for RN

R06: 1st reg. in block
R07: # registers in block

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode:
not used

Angular Mode:
not used

Unused Subroutine Levels:
4

Global Labels Called:

Direct	Secondary
RN	

Local Labels In This Routine:
none

Execution Time:
1.4 seconds

Peripherals Required:
none

Interruptible? yes

Execute Anytime? no

Program File: SM

Bytes In RAM: 28

Registers To Copy: 26

Other Comments:

SK - SUSPEND KEY ASSIGNMENTS

As part of its compatibility with HP-67 operation, the HP-41 has 15 keys (top two rows plus shifted top row) which, when pressed in USER mode, will find and execute local labels (A-J and a-e). However, this feature conflicts with any global label or function assignments to these keys. Since the 41 gives precedence to the function and global label assignments, the only way to use the local label keyboard is to clear all assignments which conflict. This process is tiresome, especially if you want to switch back and forth frequently between local label and function assignments.

The PPC ROM solves this problem. **SK** suspends both global label and function key assignments, so that the local label keyboard may be used. The key assignments may be reactivated at any time by using **RK** or by reading in any program card. **SK** can be utilized from the keyboard or activated from a running program. It operates by removing and manipulating the key assignment bit-maps stored in registers ⊢ and e. The assignments that have been made remain in user memory (registers 0C0 and up), but with the key assignment bits clear the 41-C mainframe assumes that no assignments have been made. It then checks to see if the local label corresponding to a particular key is present.

Example 1: You have some functions assigned to the top row keys, but you wish to use **FI**. Just key 10, XEQ **SK** to suspend key assignments and store the bit maps in data registers 10 and 11. (Register 10 and 11 are used for **SK**, because **FI** uses registers 1 through 9.) Then GTO **FI**. When you're done with **FI** just key 10, XEQ **RK** to regain use of your key assignments.

COMPLETE INSTRUCTIONS FOR SK

1. Insert into X a register number that will define the first address of a two-register block; for storing key assignment bit maps. To use registers 10 and 11 place the number 10 in the X register.

2. XEQ **SK**; local labels are now active, key assignments inactive.

3. NOTE: If this program is single-stepped or operated with a printer in trace mode, after line 61 (x<>e) the line numbers will begin again at 01. This is due to the fact that storing all zero's into register e resets the program line counter to zero. The program will, however, still function properly.

4. If used in a program, make sure that a register number is placed in X prior to execution (as in step 2 above). Make sure the user's program does not disturb the defined 2- register block.

5. To recover use of key assignments use the sequence x, XEQ **RK**, where x is the register number used for **SK**. If for any reason the two data registers are altered, reading in a program card will reconstruct the bit maps and reactivate the key assignments. The program card may contain only deleted lines and it need not be read in USER mode.

6. **SK** saves Y, Z, and T in X, Y, and Z. L is replaced by x + 1.

TECHNICAL DETAILS

XROM: 20,04	SK	SIZE: 002

Stack Usage:
- 0 T: USED
- 1 Z: T
- 2 Y: Z
- 3 X: Y
- 4 L: USED

Alpha Register Usage:
- 5 M:
- 6 N: ALL USED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: CLEARED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: NOT USED
- 15 e: CLEARED

ΣREG: UNCHANGED

Data Registers:
- R00: TWO CONSECUTIVE REGISTERS SPECIFIED BY THE USER ARE USED
- R06: FOR STORAGE.
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Flag Usage: NONE USED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10:
- 25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 5

Global Labels Called: Direct / Secondary

Local Labels In This Routine: 14

Execution Time: .7 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	SR
Bytes In RAM:	30
Registers To Copy:	40

Other Comments:

Routine Listing For:	SK
53♦LBL "SK"	62♦LBL 14
54 SIGN	63 "*"
55 CLX	64 X<> [
56 X<> '	65 STO \
57 XEQ 14	66 ASTO IND L
58 ISG L	67 RDN
59 ""	68 RTN
60 .	
61 X<> e	

LINE BY LINE ANALYSIS OF SK

53	LBL SK	
54	SIGN	store 1st defined register in L
55	CLX	clear X
56	X<>⊦	bring ⊦ (unshifted key bits) into X, store zeros in ⊦
57	XEQ 14	go to subroutine 14 to store key bits in defined register
58	ISG L	increment L by one to define second register
59	hex F0	NOP
60	.	enter zero into X
61	X<>e	bring e (shifted key bits) into X, store zeros in e
62	LBL 14	actual storing routine
63	"*"	clear alpha and inserts a star into last byte of M
64	X<>M	key bit register now in M, X contains a star in last byte
65	STO N	alpha now contains a star + 7 byte key bit register
66	ASTO IND L	alpha store a star + first 5 bytes of key bit register
67	RDN	restore stack
68	RTN	

CONTRIBUTORS HISTORY FOR SK

SK was written by Keith Jarett (4360) in response to a suggestion by Gary M. Tenzer (1816)

FURTHER ASSISTANCE ON SK

Call Keith Jarett (4360) at (213) 374-2583.
Call Keith Kendall (5425) at (801) 967-8080.

NOTES

SM - STACK TO MEMORY

This routine is designed to store the "stack" in data memory for furture use. The "stack" is the standard HP stack of X, Y, Z, T plus last X. These five registers are stored in five contiguous data registers, the lowest register being used to define their location in memory. SM uses the number in register 06 as the first register to use to store the "stack" values. The inverse routine is MS .

Example 1: The stack values shown on the left are to be stored in memory starting with R10.

T:	TTT	10 STO 06	R10:	XXX
Z:	ZZZ	XEQ SM ⟶	R11:	YYY
Y:	YYY		R12:	ZZZ
X:	XXX	Data stored	R13:	TTT
L:	LLL	as shown at	R14:	LLL
		right		

COMPLETE INSTRUCTIONS FOR SM

SM uses register 06 to provide the lowest register of a contiguous block of five registers. Any number, n, may be used (stored in R06) provided (n ≤ SIZE) -5 and SIZE ≥ 7. The control number remains unchanged after completion. This facilitates repeated calls of SM and MS without "initializing" R06 each time. R06 must contain a positive integer. The numbers 2 thru 6, however, will cause the stack data to be stored into the counter register R06. This is not normally recommended. See Example 4 of MS .

MORE EXAMPLES OF SM

Example 2: Lisa wants to show her brother, David, how SM and MS work. She decides that a demonstration program would work best. Here is Lisa's program.

```
01  LBL "SM/MS"
02  " R06 VALUE?"
03  PROMPT
04  X = 0?
05  SF 05
06  STO 06
07  " LOAD STACK"
08  PROMPT
09  XROM SM
10  CLST
11  STO L
12  STOP
13  XROM MS
14  FS?C 05
15  RCL IND 06
16  STOP
17  GTO  SM / MS
```

Lines 02 & 03 prompt for positive integer input. Flag 5 is set if input value is zero and the value stored in register 06 in lines 04-06. Lines 07 & 08 prompt for the user to store values in the "stack". The stack is stored at line 09 and cleared by the following two lines. The stop at line 12 allows the user to verify that the "stack" is now cleared. Pressing R/S brings back the stack with line 13. Line 14 checks flag 5 and if set executes line 14. (See WARNING in MS). The DEMO is complete with line 16, but R/S causes it to repeat with the conditional branch at line 17.

David is a skeptic and decides to verify that SM and MS really work. He assigns SM to key 11 and MS to key 15. After storing 1 in R06, he alternately presses the two keys many times. It works! David next did the same thing in program and found that ten SM , MS pairs took 17.7 seconds or less than 2 seconds to store, then recover the stack.

Routine Listing For: **SM**

```
01♦LBL "SM"
02 XEQ 04
03 XEQ 04
04 XEQ 04
05 XEQ 04
06 LASTX
07 STO IND 06
08 4
09 ST- 06
10 RTN
11♦LBL 04
12 STO IND 06
13 RDN
14 1
15 ST+ 06
16 RDN
17 RTN
```

LINE BY LINE ANALYSIS OF SM

Five registers must be stored. The store operation is done by the LBL 04 routine in lines 11 thru 17. The X register is stored in the first register specified by R06. The X register is rolled down and replaced by the 1 at line 14. Register 06 is increased by 1 at line 15. The 1 is rolled down at line 16. This process is repeated four times for X, Y, Z, & T. The Last X register is recalled and stored by lines 06 and 07. R06 is n + 4 at this time and lines 08 and 09 restore R06 to its initial value.

CONTRIBUTORS HISTORY FOR SM

The four high RPN stack is a busy set of registers since nearly all operations utilize the X register and the stack. The ability to store the stack away for future use hasn't been practical until the PPC ROM. The use of a pointer register permits many "parallel" stacks to be maintained with minimum effort. SM was written by Richard Nelson (1).

FINAL REMARKS FOR SM

SM , like many housekeeping and data processing operations in the ROM, are probably best implemented in assembly language. If implemented in a future PPC EPROM the "stack" should be unaltered and simply copied to data memory.

FURTHER ASSISTANCE ON SM

Richard Nelson (1) -- (714) 754-6226 P.M.
Richard Schwartz (2289) -- (213) 447-6574 Evenings.

NOTES

TECHNICAL DETAILS

XROM: 20,55	SM	SIZE: ≥ 005*

Stack Usage:

0 T: USED
1 Z: USED
2 Y: USED
3 X: USED
4 L: USED

Alpha Register Usage:

5 M: NOT USED
6 N: NOT USED
7 O: NOT USED
8 P: NOT USED

Other Status Registers:

9 Q:
10 ⊢:
11 a: NONE USED BY
12 b: ROUTINE
13 c:
14 d:
15 e:

ΣREG: NOT USED

Data Registers:

R00:

R06: Lowest Reg. of block.
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: NONE

04:
05:
06:
07:
08:
09:
10:

25:

Display Mode: N/A

Angular Mode: N/A

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine:

NONE

Execution Time: 1.3 seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: SM
Bytes In RAM: 36
Registers To Copy: 26

Other Comments:

*Minimum size is value in R06 + 5.

SR - SHORTEN RETURN STACK

SR is the companion routine to LR (lengthen return stack). The LR / SR combination permits the use of more than six subroutine levels. LR and SR are simple to use, as shown in LR Example 1. Furthermore, LR Application Programs 1 and 2 make it easy to use LR and SR to construct recursive programs.

LR stores five subroutine return addresses in a pair of data register, allowing the user program to call another six levels. After returning, the user program calls SR to extract the return addresses from the pair of data registers and place them in status registers a and b, which contain the return stack. The data registers are not altered and the return addresses they contain may be used again if desired. This feature is used by Application Program SRS (see LR).

COMPLETE INSTRUCTIONS FOR SR

Place the number of the first of a pair of data registers in X (the same number used for LR) and XEQ SR . This takes the 5 return pointers out of the status registers and puts them into the status registers as the second through sixth return address. These become the first through fifth returns when SR returns to the calling program.

See the LR write-up for a typical use of SR (LR Example 1), supportive programs which use LR and SR to manage a variable-length artificial return stack, and two examples of recursive (self-calling) programs.

LINE BY LINE ANALYSIS OF SR

The accompanying stack and alpha analysis form for SR fully explains the workings of the program. The notation used is described in the LR write-up.

CONTRIBUTORS HISTORY FOR SR

See LR .

FURTHER ASSISTANCE ON SR

Call Paul Lind (6157) at (206) 525-1033.
Call Harry Bertuccelli (3994) at (213) 846-6390.

Routine Listing For: SR

```
01+LBL "SR"          14 X<> IND L
02 SIGN              15 STO \
03 SF 10             16 "⊢**"
04 RDN               17 X<> IND L
05 RCL b             18 STO ]
06 STO [             19 "⊢**"
07 RDN               20 X<> ]
08 FC?C 10           21 STO a
09 RTN               22 X<> \
10 "⊢***"            23 CLA
11 RCL IND L
12 ISG L             24+LBL "Sb"
13 ""                25 STO b
```

TECHNICAL DETAILS

XROM: 20,00 | SR | SIZE: 002

Stack Usage:
- 0 T: USED
- 1 Z: T
- 2 Y: Z
- 3 X: Y
- 4 L: X + 1

Alpha Register Usage:
- 5 M: TEMPORARY b
- 6 N: CLEARED
- 7 O: CLEARED
- 8 P: CLEARED

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: RETURN POINTERS
- 12 b: ARE ALTERED
- 13 c: NOT USED
- 14 d: NOT USED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers:
- R00: TWO CONSECUTIVE REGISTERS SPECIFIED BY THE USER ARE
- R06: RECALLED
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Flag Usage: ONLY FLAG 10 IS USED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10: CLEARED
- 25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels:
0 CALLED BY A PROGRAM
1 FROM THE KEYBOARD

Global Labels Called:

Direct	Secondary
Sb (IN LINE)	NONE

Local Labels In This Routine:
NONE

Execution Time: .9 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	SR
Bytes In RAM:	59
Registers To Copy:	60

Other Comments:

STACK AND ALPHA REGISTER ANALYSIS FOR **SR**

T	Z	Y	X	L	P	O	N	M	L-# INSTRUCTION
T_0	Z_0	Y_0	n						01♦LBL "SR"
T_0	Z_0	Y_0	1	n					02 SIGN
T_0	Z_0	Y_0	1	n					03 SF 10
1	T_0	Z_0	Y_0	n					04 RDN
T_0	Z_0	Y_0	32211PP	n					05 RCL b
T_0	Z_0	Y_0	32211PP	n				32211PP	06 STO [
32211PP	T_0	Z_0	Y_0	n				32211PP	07 RDN
32211PP	T_0	Z_0	Y_0	n				32211PP	08 FC?C 10
32211PP	T_0	Z_0	Y_0	n				32211PP	09 RTN
32211PP	T_0	Z_0	Y_0	n			322	11PP***	10 "├***"
T_0	Z_0	Y_0	"66554"	n			322	11PP***	11 RCL IND L
T_0	Z_0	Y_0	"66554"	n+1			322	11PP***	12 ISG L
T_0	Z_0	Y_0	"66554"	n+1			322	11PP***	13 ""
T_0	Z_0	Y_0	"43322"	n+1			322	11PP***	14 X<> IND L
T_0	Z_0	Y_0	"43322"	n+1			+43322	11PP***	15 STO \
T_0	Z_0	Y_0	"43322"	n+1		+	4332211	PP*****	16 "├**"
T_0	Z_0	Y_0	"+66554"	n+1		+	4332211	PP*****	17 X<> IND L
T_0	Z_0	Y_0	"+66554"	n+1		+66554	4332211	PP*****	18 STO]
T_0	Z_0	Y_0	"+66554"	n+1	+	6655443	32211PP	*******	19 "├**"
T_0	Z_0	Y_0	6655443	n+1		+66554	32211PP	*******	20 X<>]
T_0	Z_0	Y_0	6655443	n+1		+66554	32211PP	*******	21 STO a
T_0	Z_0	Y_0	32211PP	n+1		+66554	6655443	*******	22 X<> \
T_0	Z_0	Y_0	32211PP	n+1					23 CLA
T_0	Z_0	Y_0	32211PP	n+1					24♦LBL "Sb"
T_0	Z_0	Y_0	32211PP	n+1					25 STO b
T_0	Z_0	Y_0	32211PP	n+1				32211PP	06 STO [
32211PP	T_0	Z_0	Y_0	n+1				32211PP	07 RDN
32211PP	T_0	Z_0	Y_0	n+1				32211PP	08 FC?C 10
32211PP	T_0	Z_0	Y_0	n+1				32211PP	09 RTN

SU - SUBSTITUTE CHARACTER

SU is an alpha register editing subroutine. SU replaces one of the ten rightmost bytes with the rightmost byte in the Y register. The integer part of the number in the X register, one through ten, designates the position number of the byte to be replaced. Bytes are counted from right to left. The data in Y may be of any type, but the alpha register should contain no more than thirteen characters. SU preserves the integer part of X (position number) in LASTX. The replacement data in Y, and the user data in Z, are preserved in X and Y, respectively. SU makes no subroutine calls, uses no data registers and executes in 1.7 seconds. Flag 25 is cleared, but all other states are preserved.

Example 1: Byte Extraction and Replacement

Change the alpha string I LOVE to I LIVE, using three different methods to generate the replacement I.

Method 1: Alpha Character

DO:	SEE:	RESULT:
AON, I	I_	replacement I
ASTO X	I	stored in X
I LOVE	I LOVE_	alpha string
AOFF, 3	3_	position # of O
XEQ SU	I	Y&Z drop to X&Y
AON	I LIVE	byte replaced

Method 2: Exponent

DO:	SEE:	RESULT:
AON, I LOVE	I LOVE_	alpha string
AOFF, EEX 49	1 _49	hex address of I
ENTER, 3	3	position # of O
XEQ SU	1.0000 49	Y&Z drop to X&Y
AON	I LIVE	byte replaced

Method 3: Byte Extraction

DO:	SEE:	RESULT:
AON, I LOVE	I LOVE_	alpha string
ASTO X, AOFF	I LOVE	stored in X
6	6_	position # of I
XEQ NC	I	byte extracted
AON, CLA		clear ALPHA
ARCL Y	I LOVE_	original string
AOFF, 3	3	position # of O
XEQ SU	I	Y&Z drop to X&Y
AON	I LIVE	byte replaced

Example 2: Delete Leftmost Byte

DO:	SEE:	RESULT:
AON, ABCDEF	ABCDEF_	alpha string
AOFF, 0	0_	replacement NULL
ENTER, 6	6_	number of bytes
XEQ SU	0.0000	Y&Z drop to X&Y
AON	BCDEF	byte deleted

COMPLETE INSTRUCTIONS FOR SU

SU replaces one of the ten rightmost bytes, and preserves a string of up to 13 characters in the alpha register. SU deletes the leftmost of up to ten bytes; the number of bytes in ALPHA (excluding leading NULLs) must be known.

A combined hex table is useful reference for SU editing. This reference is used to determine the address of the replacement byte, and indicates which 100 out of the 256 bytes can be constructed using an exponent.

To use SU to replace a byte, place the replacement byte in the rightmost position in Y, and the data to be altered in ALPHA. Numbers and non-normalized data are stored and recalled from ALPHA using such instructions as STO M, RCL M and, X<>M. Then, place a number (one through ten) in X. The integer part of the number represents the position number of the byte to be replaced. Bytes are counted from right to left and two digits constitute one byte. Then, XEQ left. Then, XEQ "SU" and the byte is replaced.

To use SU to delete the leftmost of up to ten bytes, place a NULL in the rightmost position in Y. This Null may be a zero exponent or an alpha NULL. The number zero has, of course, a zero exponent, as do numbers of which the absolute value is greater than or equal to one and less than ten. The alpha NULL is created using CLA, ASTO Y, giving a blank display. The alpha NULL is suppressed as a leading byte, and otherwise displays as an overline in ALPHA: overlines are not displayed in the stack. To continue, with the data to be altered in ALPHA, place a number (one through ten) in X. The integer part of the number should equal the number of bytes in ALPHA (excluding leading NULLs). Then, XEQ SU and the leftmost byte is deleted.

SU can be used with strings of more than 13 characters in ALPHA if X ≠ 1, i.e., if the rightmost character is not the one being replaced. Under these conditions the 14 rightmost characters (except the one being replaced) will be preserved. If X = 1 the 14th character is usually not preserved. A fast alternate method to replace the rightmost character is to XEQ AD and then append or ARCL the desired character.

The integer part of X is preserved in LASTX, the replacement data in Y is dropped to X, and the user data previously in Z is preserved in Y. SU leaves a copy of the current contents of M in T, and the current contents of d are left in Z. SU makes no subroutine calls, uses no data registers and executes in 1.7 seconds. Flag 25 is cleared. SU may be SST'ed with one precaution: at the X<>P instruction at line 215, the display must be in SCI 0 mode with Flags 28 and 29 clear. This is because the display status is encoded in the leftmost byte of register P during the default run mode (as well as during a PSE or VIEW in a running program). This encoding is not performed when SU is executed as a running program. Lastly, do not use the printer instruction PRSTK when SST'ing due to normalization of the stack.

APPLICATION PROGRAM 1 FOR SU

SU is easily altered to insert a byte to the left of one of the nine rightmost bytes. Flag 24 is used to control the replacement/insertion option. Clear Flag 24 to select the replacement mode, and set Flag 24 to enable the insertion mode.

Four instructions are inserted in the NC / SU subroutines. First, CF 24 is inserted immediately following LBL NC. Next, insert FC? 24 immediately preceding the ⊦* instruction (line 190) and again immediately preceding the DSE L instruction (line 202). Finally, FC?C 24 is inserted immediately preceding the ISG L instruction (line 213).

APPLICATION PROGRAM 2 FOR SU

This alternate version of SU was written by Richard Chandler (6152). It arrived during the ROM loading. It is remarkable in that it contains no synthetic instructions. This SU is another version for the xth character (1 ≤ X ≤ 12) from the left of a string of up to 12 characters (if the string is longer only the leftmost 12 characters are saved). The counting of characters from the left may be more useful for some applications. The former Y (a single alpha character for the substitution) is left in X, but the rest of the stack is lost and the display format is usually changed.

APPLICATION PROGRAM FOR: SU

```
01♦LBL "NC"
02 6
03 X<Y?
04 GTO 10
05 ASTO L
06 CLA
07 ARCL L
08 RDN
09 GTO 11

10♦LBL 10
11 ASHF
12 -

13♦LBL 11
14 9
15 -
16 XEQ 14
17 ASHF
18 ASTO L
19 CLA
20 "*"
21 ARCL L
22 ASHF
23 RDN
24 RTN

25♦LBL "SU"
26 6
27 X<Y?
28 GTO 12
29 RDN
30 ASTO T
31 ASHF
32 ASTO Z
33 CLA
34 ARCL T
35 XEQ 13
36 ARCL Y
37 RTN

38♦LBL 12
39 -
40 ASTO Z
41 ASHF
42 XEQ 13
43 ASTO L
44 CLA
45 ARCL Y
46 ARCL L
47 RTN

48♦LBL 13
49 9
50 -
51 XEQ 14
52 12
53 +
54 ASHF
55 ASTO T
56 ASHF
57 ASTO L
58 CLA
59 "*"
60 ARCL T
61 ASTO T
62 CLA
63 ARCL T
64 XEQ 14
65 ASHF
66 ASHF
67 ARCL Y
68 ARCL L
69 RDN
70 RTN

71♦LBL 14
72 FIX IND X
73 ASTO T
74 CLA
75 ARCL X
76 ARCL T
77 .END.
```

LINE BY LINE ANALYSIS OF SU

SU and NC use a common program lines and use Flag 25 to skip lines and branch as required. Flag 25 is initially set, and cleared at line 191. Line 178 returns the integer part of X for the position number, which is subtracted from ten and simultaneously stored in LASTX at line 181. The difference is used to indirectly set the number of display digits for the ARCL sequence, lines 182 through 186. Line 184 appends a variable number of bytes, from 4 through 13, to the original string in ALPHA: these bytes will be deleted in lines 202 through 219. A position number of one will append 13 bytes and a ten will append 4 bytes. The result is that all the bytes to the left of the byte to be replaced are now in P and O, and the byte to be replaced is in the leftmost position in N. The RCL d and STO d instructions preserve the original display status; these instructions perform the same function at lines 206 and 210.

Lines 187 and 189 store the bytes to the left of the byte to be replaced in the stack. Line 190 shifts the byte to be replaced into the rightmost position in O. Line 194 stores the replacement byte previously in Y into O. Line 195 fills M with stars, which are deleted at line 201. Lines 196 through 199 restore the bytes to the left of the replaced byte back into P and O; lines 199 through 201 restore the data, with the byte replaced, to the position assumed after execution of line 184.

REFERENCES FOR SU

Wickes, William C. "Synthetic Function Routines." *PPC CALCULATOR JOURNAL*, V7N3P7 (April 1980).

Wickes, William C. *SYNTHETIC PROGRAMMING ON THE HP-41C*. (Larken Publications, 1980), page 64.

CONTRIBUTORS HISTORY FOR SU

William C. Wickes (3735) presented one of the first practical applications of synthetic programming with his SUB, the original alpha-manipulation routine. HM, his program version of "Hangman," the familiar word-guessing game, effectively demonstrates a use of SUB.

Carter P. Buck (4783) wrote the SU / NC integrated subroutines of the PPC Custom ROM. These are improved versions of Wickes's SUB and ISO.

FURTHER ASSISTANCE ON SU

Call Carter Buck (4783) at (415) 653-6901.
Call Richard Chandler (6152) at (919) 851-2153.

Routine Listing For: SU

```
175♦LBL "SU"
176 SF 25

177♦LBL 14
178 INT
179 E1
180 X<>Y
181 -
182 RCL d
183 SCI IND Y
184 ARCL Y
185 STO d
186 RDN
187 X<> ]
188 FS? 25
189 RCL ↑
190 "⊦*"
191 FC?C 25
192 GTO 14
193 X<> Z
194 STO ]
195 "⊦******"
196 X<> Z
197 STO ↑
198 RDN
199 X<> ]
200 X<> \
201 STO [
202 DSE L
203 CLX
204 X<> L
205 10↑X
206 RCL d
207 FIX 0
208 CF 29
209 ARCL Y
210 STO d
211 RDN
212 CLX
213 ISG L
214 CLX
215 X<> ↑
216 STO \
217 CLX
218 X<> ]
219 STO [
220 RDN
221 RTN

222♦LBL 14
223 X<> ]
224 CLA
225 STO [
226 ASTO X
227 END
```

TECHNICAL DETAILS

XROM: 10,39 | **SU** | SIZE: 000

Stack Usage:
- 0 T: new M
- 1 Z: d
- 2 Y: Z
- 3 X: Y
- 4 L: X

Flag Usage: ONLY FLAG 25 USED
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10:
- 25: CLEARED

Alpha Register Usage:
- 5 M:
- 6 N: NONE ALTERED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R06:
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine:
14 (TWICE)

Execution Time: 1.5 seconds.

Peripherals Required: NONE

Interruptible? YES

Execute Anytime? NO

Program File: VK

Bytes In RAM: 102 WITH END

Registers To Copy: 63

Other Comments:

NOTES

APPENDIX K - ROUTINE LABEL-XROM TABLE

WAND FUNCTIONS & XROM NUMBERS

XROM 27,01	WNDDTA
XROM 27,02	WNDDTX
XROM 27,03	WNDLNK
XROM 27,04	WNDSUB
XROM 27,05	WNDSCN
XROM 27,06	XROM "WNDTST"

PRINTER FUNCTIONS & XROM NUMBERS

XROM 29,01	ACA
XROM 29,02	ACCHR
XROM 29,03	ACCOL
XROM 29,04	ACSPEC
XROM 29,05	ACX
XROM 29,06	BLDSPEC
XROM 29,07	LIST
XROM 29,08	PRA
XROM 29,09	XROM "PRAXIS"
XROM 29,10	PRBUF
XROM 29,11	PRFLAGS
XROM 29,12	PRKEYS
XROM 29,13	PRP
XROM 29,14	XROM "PRPLOT"
XROM 29,15	XROM "PRPLOTP"
XROM 29,16	PRREG
XROM 29,17	PRREGX
XROM 29,18	PRΣ
XROM 29,19	PRSTK
XROM 29,20	PRX
XROM 29,21	REGPLOT
XROM 29,22	SKPCHR
XROM 29,23	SKPCOL
XROM 29,24	STKPLOT

CARD READER FUNCTIONS & XROM NUMBERS

XROM 30,01	MRG
XROM 30,02	RDTA
XROM 30,03	RDTAX
XROM 30,04	RSUB
XROM 30,05	VER
XROM 30,06	WALL
XROM 30,07	WDTA
XROM 30,08	WDTAX
XROM 30,09	WPRV
XROM 30,10	WSTS
XROM 30,11	7CLREG
XROM 30,12	7DSP0
XROM 30,13	7DSP1
XROM 30,14	7DSP2
XROM 30,15	7DSP3
XROM 30,16	7DSP4
XROM 30,17	7DSP5
XROM 30,18	7DSP6
XROM 30,19	7DSP7
XROM 30,20	7DSP8
XROM 30,21	7DSP9
XROM 30,22	7DSPI
XROM 30,23	7DSZ
XROM 30,24	7DSZI
XROM 30,25	7ENG
XROM 30,26	7FIX
XROM 30,27	7GSBI
XROM 30,28	7GTOI
XROM 30,29	7ISZ
XROM 30,30	7ISZI
XROM 30,31	7P<>S
XROM 30,32	7PRREG
XROM 30,33	7PRSTK
XROM 30,34	7PRTX
XROM 30,35	7RCLΣ
XROM 30,36	7SCI

GLOBAL LABEL TO XROM NUMBER

"+K"	XROM 10,03	"LR"	XROM 20,02
"-B"	XROM 10,24	"M1"	XROM 20,33
"1K"	XROM 10,02	"M2"	XROM 20,31
"2D"	XROM 10,55	"M3"	XROM 20,32
"A?"	XROM 10,10	"M4"	XROM 20,35
"AD"	XROM 10,18	"M5"	XROM 20,36
"AL"	XROM 10,37	"MA"	XROM 20,54
"AM"	XROM 20,53	"MK"	XROM 10,01
"Ab"	XROM 10,61	"ML"	XROM 10,12
"BA"	XROM 20,30	"MP"	XROM 20,28
"BC"	XROM 20,43	"MS"	XROM 10,48
"BD"	XROM 20,17	"MT"	XROM 10,28
"BE"	XROM 20,34	"NC"	XROM 10,38
"BI"	XROM 10,44	"NH"	XROM 10,40
"BL"	XROM 10,42	"NP"	XROM 20,14
"BM"	XROM 20,39	"NR"	XROM 20,50
"BR"	XROM 20,40	"NS"	XROM 20,49
"BV"	XROM 20,07	"OM"	XROM 10,58
"BX"	XROM 20,41	"PA"	XROM 10,59
"BΣ"	XROM 20,42	"PD"	XROM 10,52
"C?"	XROM 10,16	"PK"	XROM 10,09
"CA"	XROM 20,23	"PM"	XROM 20,19
"CB"	XROM 10,50	"PR"	XROM 20,45
"CD"	XROM 10,35	"PO"	XROM 20,51
"CJ"	XROM 20,21	"PS"	XROM 10,46
"CK"	XROM 10,06	"QR"	XROM 10,54
"CM"	XROM 20,20	"RD"	XROM 20,05
"CP"	XROM 20,27	"RF"	XROM 10,13
"CU"	XROM 10,34	"RK"	XROM 20,06
"CV"	XROM 20,08	"RN"	XROM 20,16
"CX"	XROM 10,33	"RT"	XROM 10,51
"DC"	XROM 10,11	"RX"	XROM 10,57
"DF"	XROM 20,13	"Rb"	XROM 20,52
"DP"	XROM 10,53	"S1"	XROM 20,46
"DR"	XROM 20,38	"S2"	XROM 20,48
"DS"	XROM 10,29	"S3"	XROM 20,47
"DT"	XROM 10,17	"S?"	XROM 10,15
"E?"	XROM 10,62	"SD"	XROM 20,03
"EP"	XROM 10,31	"SE"	XROM 20,56
"EX"	XROM 10,27	"SK"	XROM 20,04
"F?"	XROM 10,04	"SM"	XROM 20,55
"FD"	XROM 20,11	"SR"	XROM 20,00
"FI"	XROM 10,63	"SU"	XROM 10,39
"FL"	XROM 10,43	"SV"	XROM 20,10
"FR"	XROM 20,12	"SX"	XROM 10,56
"GE"	XROM 10,60	"Sb"	XROM 20,01
"GN"	XROM 20,15	"T1"	XROM 10,47
"HA"	XROM 20,25	"TB"	XROM 20,18
"HD"	XROM 10,20	"TN"	XROM 10,32
"HN"	XROM 10,41	"UD"	XROM 10,08
"HP"	XROM 20,29	"UR"	XROM 20,44
"HS"	XROM 20,26	"VA"	XROM 10,07
"IF"	XROM 10,49	"VF"	XROM 20,58
"IG"	XROM 20,09	"VK"	XROM 10,36
"IP"	XROM 10,45	"VM"	XROM 10,26
"IR"	XROM 20,37	"VS"	XROM 10,30
"JC"	XROM 20,22	"XD"	XROM 10,25
"L-"	XROM 10,23	"XE"	XROM 10,19
"LB"	XROM 10,22	"XL"	XROM 20,57
"LF"	XROM 10,05	"Σ?"	XROM 10,14
"LG"	XROM 20,24	"ΣC"	XROM 10,21

XROM NUMBER TO GLOBAL LABEL

XROM 10,01	"MK"	XROM 10,62	"E?"
XROM 10,02	"1K"	XROM 10,63	"FI"
XROM 10,03	"+K"	XROM 20,00	"SR"
XROM 10,04	"F?"	XROM 20,01	"Sb"
XROM 10,05	"LF"	XROM 20,02	"LR"
XROM 10,06	"CK"	XROM 20,03	"SD"
XROM 10,07	"VA"	XROM 20,04	"SK"
XROM 10,08	"UD"	XROM 20,05	"RD"
XROM 10,09	"PK"	XROM 20,06	"RK"
XROM 10,10	"A?"	XROM 20,07	"BV"
XROM 10,11	"DC"	XROM 20,08	"CV"
XROM 10,12	"ML"	XROM 20,09	"IG"
XROM 10,13	"RF"	XROM 20,10	"SV"
XROM 10,14	"Σ?"	XROM 20,11	"FD"
XROM 10,15	"S?"	XROM 20,12	"FR"
XROM 10,16	"C?"	XROM 20,13	"DF"
XROM 10,17	"DT"	XROM 20,14	"NP"
XROM 10,18	"AD"	XROM 20,15	"GN"
XROM 10,19	"XE"	XROM 20,16	"RN"
XROM 10,20	"HD"	XROM 20,17	"BD"
XROM 10,21	"ΣC"	XROM 20,18	"TB"
XROM 10,22	"LB"	XROM 20,19	"PM"
XROM 10,23	"L-"	XROM 20,20	"CM"
XROM 10,24	"-B"	XROM 20,21	"CJ"
XROM 10,25	"XD"	XROM 20,22	"JC"
XROM 10,26	"VM"	XROM 20,23	"CA"
XROM 10,27	"EX"	XROM 20,24	"LG"
XROM 10,28	"MT"	XROM 20,25	"HA"
XROM 10,29	"DS"	XROM 20,26	"HS"
XROM 10,30	"VS"	XROM 20,27	"CP"
XROM 10,31	"EP"	XROM 20,28	"MP"
XROM 10,32	"TN"	XROM 20,29	"HP"
XROM 10,33	"CX"	XROM 20,30	"BA"
XROM 10,34	"CU"	XROM 20,31	"M2"
XROM 10,35	"CD"	XROM 20,32	"M3"
XROM 10,36	"VK"	XROM 20,33	"M1"
XROM 10,37	"AL"	XROM 20,34	"BE"
XROM 10,38	"NC"	XROM 20,35	"M4"
XROM 10,39	"SU"	XROM 20,36	"M5"
XROM 10,40	"NH"	XROM 20,37	"IR"
XROM 10,41	"HN"	XROM 20,38	"DR"
XROM 10,42	"BL"	XROM 20,39	"BM"
XROM 10,43	"FL"	XROM 20,40	"BR"
XROM 10,44	"BI"	XROM 20,41	"BX"
XROM 10,45	"IP"	XROM 20,42	"BΣ"
XROM 10,46	"PS"	XROM 20,43	"BC"
XROM 10,47	"T1"	XROM 20,44	"UR"
XROM 10,48	"MS"	XROM 20,45	"PR"
XROM 10,49	"IF"	XROM 20,46	"S1"
XROM 10,50	"CB"	XROM 20,47	"S3"
XROM 10,51	"RT"	XROM 20,48	"S2"
XROM 10,52	"PD"	XROM 20,49	"NS"
XROM 10,53	"DP"	XROM 20,50	"NR"
XROM 10,54	"QR"	XROM 20,51	"PO"
XROM 10,55	"2D"	XROM 20,52	"Rb"
XROM 10,56	"SX"	XROM 20,53	"AM"
XROM 10,57	"RX"	XROM 20,54	"MA"
XROM 10,58	"OM"	XROM 20,55	"SM"
XROM 10,59	"PA"	XROM 20,56	"SE"
XROM 10,60	"GE"	XROM 20,57	"XL"
XROM 10,61	"Ab"	XROM 20,58	"VF"

SV - SOLVE ROUTINE

This routine is a simple root solving program which will approximate a solution to an equation of the form: f(x)=0 using the Secant Method (a simplified form of Newton's Method). SV will find only one root at a time. The program requires an initial guess and an initial step size. The output is an x value which most closely makes f(x)=0. A flag may be set to display the successive approximations as they converge to the final answer. Convergence depends on the initial guess. Accuracy depends on the display setting.

Example 1: Use SV to find the two roots of the quadratic equation $x^2 + 2*x - 15 = 0$.

1. Insure a minimum SIZE 010.
2. Select a display setting of SCI 5. The routine will end when two successive approximations are rounded and found to be equal according to the display setting.
3. Set flag F10 to view the successive approximations.
4. The function on the left side of the equation must be programmed as a subroutine. The input to this subroutine, namely x, is assumed to be in the X-register and can be recalled from R07. The output from this subroutine, namely f(x), is also to be left in the X-register. For this example the following routine may be programmed in RAM program memory.

```
01*LBL "FX1"
02 X↑2
03 LASTX
04 2
05 *
06 +
07 15
08 -
09 RTN
```

5. The name of the global label "FX1" should be stored in R06. Go into alpha mode and key "FX1" ASTO 06.
6. The initial guess (nonzero) is to be entered along with an initial step size which may be zero or may be a small number compared to x. If a 0 step size is entered then the program will calculate the first step as 1% of the intial guess x. The program will also accept a non-zero value as the intial step size. For most applications and for this example use 0 as the initial step size. Choose x=7 as the initial guess for x. Key 0 ENTER↑ 7
7. XEQ "SV". The following consecutive approximations will be displayed.

```
7.00000+00
6.93000+00
3.98682+00
3.30024+00
3.03190+00
3.00115+00
3.00000+00
```

The final solution is returned after about 8 seconds. The true answer is exactly x=3. Since the above quadratic has two roots we will key in another initial guess to search for the other root.
This time we will guess x = -10. Key 0 ENTER↑ 10 CHS and XEQ "SV". The following sequence of numbers will be displayed.

```
-1.00000+01
-9.90000+00
-6.36872+00
-5.47003+00
-5.06539+00
-5.00360+00
-5.00003+00
-5.00000+00
```

The true answer this time is exactly x = -5.

COMPLETE INSTRUCTIONS FOR SV

(Keyboard Operations):

To calculate a root of f(x)=0:

1) Select SIZE. The minimum size required by SV is SIZE 010. The storage requirements for constants and coefficients associated with the function f(x) may dictate a larger size.

2) Select display setting. The display setting will generally determine when SV ends. If an exact solution is found then SV will end on the next iteration, otherwise SV rounds the last two approximations and ends if those rounded values are equal. In general, a display setting of SCI n will produce (optimistically) a solution correctly rounded to n+1 significant figures. A display mode of SCI or ENG is generally preferred to a FIX mode.

3) Specify display option. Flag 10 controls a display option. If F10 is set then the successively calculated approximations will be displayed. In this manner the user may view the progress of the iterations. This is especially recommended in the SV routine since SV may fail to converge if the initial guess is too far away from an actual root. Even when the values stabilize they may oscillate and it is a simple matter for the user to manually stop the program. If F10 is clear only the final x-value is returned.

4) Program the function f(x)=0. The function f(x) represented by one side of the equation must be programmed as a subroutine in program memory which starts with a global label name and ends with a RTN or END instruction. The label name should be of six or less characters and should be stored in R06. The input x and the output f(x) are both assumed to be in the X-register. The input x may also be recalled from R07. Since global label search begins from the bottom of program memory, it is advisable to place f(x) near the bottom of program memory. The f(x) program should not use registers R06-R09 and should not disturb flag F10.

5) Store the global label name from step 4) (six or less characters) in R06. The function subroutine call will be made via an XEQ IND 06 instruction.

6) Specify initial step size and initial guess. SV requires two input values. The first input is the step size which the program uses to determine the approximation for the derivative at the initial guess. The second input is the initial guess and is used as the starting x value by the program. The closer the initial guess is to the true solution the quicker the solution is found. Do not use 0 as an initial guess.

SV

If zero is entered as the initial step size then the program will automatically calculate 1% of x as the acutal step size. A zero step size should prove adequate for the majority of applications. However, the user may enter a non-zero step size which may be finer or coarser than 1% of the initial x.

These two values are keyed in as:

step size ENTER↑ guess

7. XEQ " SV ". If F10 is set the program will display the consecutive approximations. If a printer is plugged in and turned on these approximations will be printed. The final solution will be left in the X-register when the program ends.

MORE EXAMPLES OF SV

Example 2: Solve f(x) = x^3 - x - 1 = 0.

1) SIZE 010 minimum
2) Set display mode as SCI 6.
3) Set flag F10 to view the approximations.
4) Key the following routine for f(x) into program memory:

```
LBL*FX2
ENTER↑
X↑2
1
-
*
1
-
RTN
```

5) Key "FX2" in the alpha register and press ASTO 06.
6) Key in the initial step size as 0 and key in the initial guess as x=4. Key 0 ENTER↑ 4.
7) XEQ " SV ".

The following sequence of approximations will be displayed:

```
4.000000+00
3.960000+00
2.731772+00
2.226515+00
1.780222+00
1.522190+00
1.382556+00
1.333776+00
1.325185+00
1.324722+00
1.324718+00
```

The final solution is returned after about 13 seconds. The solution is correct to the digits displayed.

Example 3: Solve f(x) = x^3 - 3*x^2 + 4 = 0.

1) SIZE 010 minimum
2) Set display mode as SCI 4.
3) Set flag F10 to view the approximations.
4) Key the following routine for f(x) into program memory:

```
LBL*FX3
X↑2
LAST X
3
-
*
4
+
RTN
```

5) Key "FX3" in the alpha register and ASTO 06.
6) Key in an initial step size of 0 and an initial guess of -2. Key 0 ENTER↑ 2 CHS
7) XEQ " SV ".

The following approximations will be displayed:

```
-2.0000+00
-1.9800+00
-1.3283+00
-1.1289+00
-1.0229+00
-1.0018+00
-1.0000+00
```

The final answer is returned after about 8 seconds. The true solution is exactly x = -1.

The following examples contain abbreviated instructions.

Example 4: The following equation is known as Kepler's equation:

x - E*SIN(x) - m = 0

and plays an important role in astronomy and astrodynamics (space travel). It can be programmed as follows:

```
LBL*FX4
ENTER↑
SIN
RCL 01        (Note: R01=E)
*
-
RCL 02        (Note: R02=m)
-
RTN
```

Set RADIANS angle mode. The equation can now be solved for any values of E (R01) and m (R02). When E=0.2 and m=0.8 the function has only one root (9.64334-01) which can be found with any initial guess.

Example 5: SV can be used to find maxima and minima of a function by solving for zeros of its derivative. For example, if f(x) = sin(x)/x then the derivative f'(x) = [x*cos(x) - sin(x)]/x^2
Zeros of f' occur where the numerator is 0. Consequently, solutions can be found by applying SV to the following function which represents the numerator g(x) = x*cos(x) - sin(x)

```
LBL*GX5
ENTER↑
COS
*
RCL 07
SIN
-
RTN
```

Assuming SCI 6 display mode and RADIANS angle mode. Store "GX5" in R06. Key in small initial guesses using a step size of 0 and SV will find the first few roots as 0, ±4.49341+00, ±7.72525+00. (Note that instead of taking the derivative algebraically, the ROM routine FD might be used).

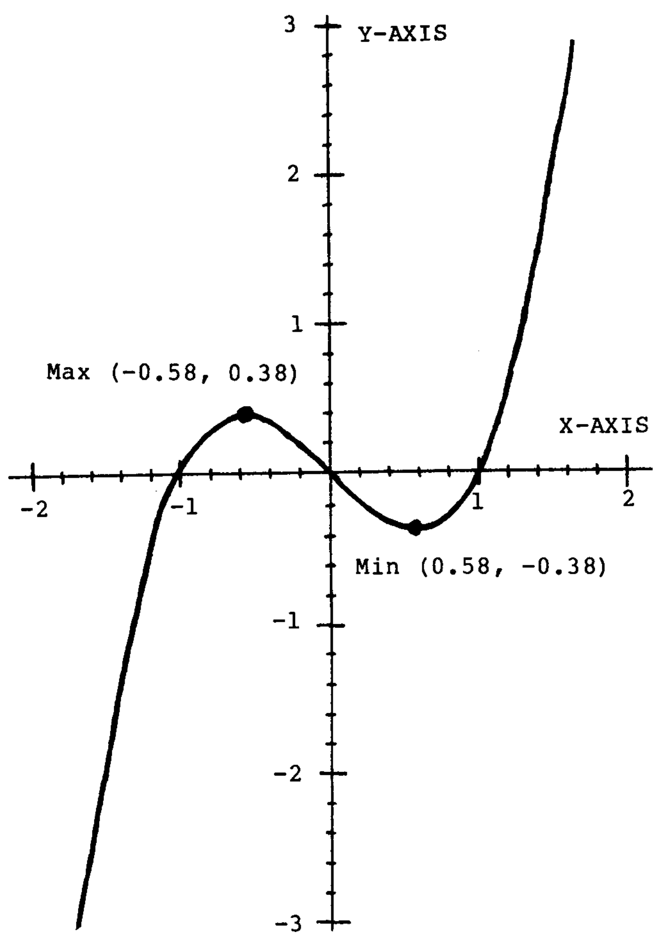
Example 6: Use **SV** to find all three roots of the cubic equation:

$$x*(x^2-1) = 0$$

The following should be programmed:

```
LBL*FX6
ENTER↑
X↑2
1
-
*
RTN
```

A sketch of the graph of this function will prove useful in understanding how the inital guesses determine which root is found.



GRAPH OF $Y = X^3 - X = X(X^2 - 1) = X(X+1)(X-1)$

The initial step size is 0 for each of the guesses suggested below. An initial guess greater than 0.6 for this example will find the root x=1.0, while a guess between -0.49 and +0.49 will find the root at x=0.0. However, guesses too close to the local peaks of the function at $x=\pm 1/\sqrt{3}$, where the slope of the tangent line is zero, lead to oscillations that fail to converge. Try an initial guess of x=0.58 to observe this behavior. This example also illustrates that the root found is not necessarily the one nearest to the initial guess. Try x=0.52 which finds the root x = -1.0.

Example 7: f(x) = x*LN(x) - 1.2 = 0

Display mode: SCI 6
Initial step size = 0
Initial guess = 3
See:

3.000000+00
2.970000+00
1.998929+00
1.902018+00
1.888327+00
1.888087+00

Result: 1.888087 after about 8 seconds

Example 8: f(x) = 3*x - COS(x) - 1 = 0

Display mode: SCI 6
Use RADIANS angle mode
Initial step size = 0
Initial guess = 2
See:

2.000000+00
1.980000+00
6.159990-01
6.078243-01
6.071024-01
6.071016-01

Result: 6.071016-01 after about 10 seconds

Example 9: $f(x) = x^2 + 4*SIN(x) = 0$

Display mode: SCI 6
Use RADIANS angle mode
Initial step size = 0
Initial guess = -4
See:

-4.000000+00
-3.960000+00
-2.210789+00
-2.037220+00
-1.946227+00
-1.934406+00
-1.933758+00
-1.933754+00

Result: -1.933754 after about 12 seconds

Example 10: $f(x) = x^4 - 26x^2 + 49x - 25 = 0$

Display mode: SCI 6
Initial step size = 0
Initial guess = 5
See:

5.000000+00
4.950000+00
4.310588+00
4.077156+00
3.927333+00
3.883035+00
3.876065+00
3.875777+00
3.875775+00

Result: 3.875775 after about 14 seconds

Continuing this same example for another root:
Initial step size = 0
Initial guess = -10
See:

-1.000000+01
-9.900000+00
-7.959395+00
-7.135530+00
-6.438967+00
-6.086301+00
-5.945260+00
-5.917701+00
-5.915863+00
-5.915842+00

Result: -5.915842+00 after about 14 seconds

FURTHER DISCUSSION OF SV

SV is not a sophisticated root solver and is subject to all the difficulties and error traps that confront all other root solvers. Limited space in the PPC ROM did not allow protection schemes to detect or rectify possible trouble areas. The method used, strictly speaking, is the Secant Method, however, it can be considered a form of Newton's Method where a numerical approximation is used for the derivative. A secant line is used to approximate the true tangent line. If SV fails to converge then another initial guess must be tried. SV can be effective as a subroutine in a program provided the user has knowledge of the range of appropriate values for the given function.

The display setting will help control the accuracy of the final result. When in SCI n display mode the final answer will (usually) be accurate to n+1 significant digits. However, sometimes this is not the case and the final answer will not be as accurate as the display setting would indicate. Every floating point operation in a computational process can give rise to rounding error which, once generated, may then be increased in subsequent operations.

For example, let $f(x) = x^2 - 6*x + 9$ and use SV to solve for f(x)=0. In FIX 9 the answer SV returns may be 3.000030072 which is accurate to five digits only. The true solution is a double root at x=3. Thus the display setting has not determined the accuracy in this example. This is basically caused by using only ten digits internally in the calculator, causing each and every calculation to have its solution rounded to ten digits. The root solver itself cannot then be held to ransom when the f(x) routine is affected by rounding errors.

This example highlights the action of SV when the secant line is horizontal, that is, when f(a)=f(b) where a and b are successive approximations. This situation may occur at multiple roots where the first derivative shares a root with the original function. (When the first derivative is zero the tangent line is horizontal). In general, do not select a display n value any larger than necessary. The use of SCI or ENG display modes are generally preferred to the FIX display mode. And do not blindly accept any solution given by this or any other root-solving program. Any potential real solution can be validated by applying the f(x) subroutine to see how close f(x) really is to 0.

Because the HP-34C calculator's SOLVE routine uses a similar method as SV (with many refinements), users are urged to study Reference 5. In that informative article some of the problems of root solving and a description of the mathematics of the secant method are discussed. Reference 1 provides a broad background to the subject. Further information may be found in most university and college libraries.

THE SELECTION OF A METHOD FOR SV

The oldest know method of root-solving is the Method of False Position, or Regula Falsi. Commencing with two estimates, lying on either side of the actual root, inverse linear interpolation is applied to produce a new estimate. Here, a linear function (a straight line) is used to approximate the true function f(x) over the interval of interest. This can be seen to be "reasonable" approximation so long as the interval is "small". As the two estimates must always straddle or bracket the root, convergence is always guaranteed. Of course, after calculating the new estimate, a decision has to be made as to which previous estimate is to be discarded.

The Method of False Position has a unity order of convergence, making the iteration time reasonably long. However, the solution is always obtained.

A method similar to False Position is the Secant Method. Although the mathematics of the two methods are identical, the difference lies in the fact that the Secant Method uses the approximations in strict sequence. Thus, the bracketing of the root is no longer necessary, and a secant is used to approximate the function f(x) over the interval of interest. Then, inverse linear interpolation is applied to produce the next estimate of the root. The Secant Method's order of convergence is approximately 1.62, which is higher than the Method of False Position, but convergence to the root is no longer guaranteed.

Both the Method of False Position and the Secant Method are in the class of two-point iterative methods, which, while the order of convergence is not high, nevertheless have high stability.

Another popular method derived from calculus using a Taylor Series is commonly known as the Newton-Raphson, or Newton's Method. In this method a tangent line to the function is used to determine the direction and amount of displacement to move from the current estimate to the new estimate. Newton's Method belongs to the class of one-point iterative methods. Newton's Method is the official and familiar name of tangent sliding philosophy, and has an order of convergence of two.

The mercurial properties of Newton's Method arise from its use of derivative information gathered at one point. Both the function and its derivative must be evaluated at each iteration. (This also results in a greater programming effort for two functions are really being evaluated). The time for an iteration is longer than the False Position and Secant Methods, which both require only one function evaluation per iteration. However, the convergence rate of Newton's Method is greater than both.

Comparison of Methods

A. Method of False Position.
Advantages:
1. Convergence is guaranteed.
2. Only one function evaluation is needed at each iteration.

Disadvantages:
1. Low (unity) order of convergence.
2. A decision is needed as to which estimate to discard to insure root-bracketing occurs.
3. The root-bracketing requirement prevents its use at multiple, even-order roots.

B. The Secant Method.
Advantages:
1. Medium (approx. 1.62) order of convergence.
2. Estimates are used in strict sequence, so no decision is needed on which estimate to discard.
3. Only one function evaluation is needed at each iteration.
4. Can be very stable.

Disadvantages:
1. Convergence is not always guaranteed.
2. May have difficulty at multiple even-order roots.

C. Newton's Method.
Advantages:
1. High (2.0) order of convergence.

Disadvantages:
1. Convergence is not always guaranteed.
2. Both the function and its derivative must be evaluated at each iteration, which increases the time per iteration.
3. The derivative must be known explicitly.
4. Can be very unstable.
5. Has difficulty at multiple, even-order roots, where the function and its first derivative both have the same root.

Summary

From the above comparison, the Secant Method was considered to be the optimum algorithm, and was selected for SV . It combines a reasonable rate of convergence, a (usually) stable two-point step, uses the calculated approximations in strict sequence, and does not require evaluation of the derivative, dispensing with the need to provide the derivative explicitly.

ROOT SOLVING DIFFICULTIES - A PRIMER

One of the most frequently occuring problems in scientific work is to find the values of x for an expression f(x) which will make f(x)=0. Those values of x are called the roots of the equation f(x)=0. The function may be given explicitly as, for example, a polynomial, or as a transcendental function. In rare caes it may be possible to obtain the exact roots by algebraic manipulations. In general, however, we can hope to obtain only approximations to the roots relying on some iterative computational procedure to produce those approximations.

In the year 1225 Leonardo of Pisa studied the equation:

$$f(x) = x^3 + 2*x^2 + 10*x - 20$$

and was able to produce the root of 1.368808107. Nobody knows by what method Leonardo found this value, but it was a remarkable achievement for his time.

Simply put, all we require are those values of x which will make f(x)=0. It should be easy, so why all the fuss? The basic difficulty stems from the fact that our root solving methods tend only to use the function expression to numerically evaluate f(x) and have no analytical knowledge of the function. If they did, better starting guesses and better exit criteria could be selected. More importantly, the best root solving method to use for a particular function could be selected.

Easy though root solving may seem, and when coupled with one of the popular methods, e.g., Newton's Method, Secant Method, Bisection Method, it may come as a surprise to know that the search for the perfect root solver is no less difficult than the search for the Holy Grail! For every root finding method put forward, a situation can be provided which will cause the method to fail and deny us the solution.

What can be done? A root solver is simply a computational process which uses known facts (data) to calculate a better approximation to the root. The basic difference among root solving methods is the way in which the known facts are used to calculate the improved estimate.

If we know situations that cause our root solver to fail to find the root, we can provide assistance by enhancing the basic method with strategies to detect these problem situations and allow an escape from them. How many difficulties may confront a root solver? How many strategies do we design into it? One, five or one hundred? If we limit our root solver to solve only a specific class of problems we may be able to implement some strategies to overcome the typical problems encountered by that class.

Starting Values

All root solvers require starting values whether they be entered manually as in SV or use some set of values, e.g., 1 and 10. If the starting value is not "close" to the root, our selected method may step away from the root, making the problem worse.

Therefore the accuracy of the starting values (guesses) can be seen to be as important as the selected root solving method and its inbuilt strategies. Do we have strategies for determining an approximation to the root we seek? In some cases the answer is yes.

Exit Criteria.

When the root solver has located a root, it exits the iteration process, gives us the answer and stops. How does the root solver know when it really has found a root? Our inbuilt exit criteria must apply certain tests to the known facts and assess if a root has been found. Again, situations can be provided to fool the exit tests, and cause execution to stop when really no root has been found (i.e., no root exists).

Should we have several exit tests built into the program? Under what conditions should each be invoked? If only the answers were simple!

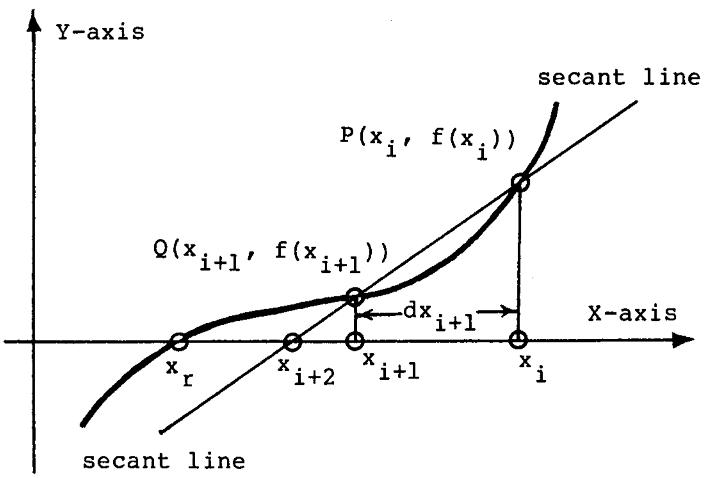
Numerical instability and round off errors.

Some root solvers can exhibit instability in certain conditions. Do we know how many of these conditions exist for any one root solver? How do we overcome these problems? As all floating point calculations are rounded to ten digits by the calculator a further difficulty arises due the propogation of these errors.

Because of the enormous difficulties which may confront a root solving process, **SV** has been written as an elementary routine, with no refinements or strategies included. It is left to the user to provide such strategies, by observing the behavior of the approximations, and taking action should divergence occur. A little experience with particular problems will provide guidance. Readers are urged to consult Reference 5.

FORMULAS USED IN SV

Let f(x) be the function whose root x_r we desire. x_{i+1} and x_i are the two previous approximations.



GRAPH OF GENERAL FUNCTION WITH SECANT LINE SHOWING APPROXIMATION X_{i+2} FOLLOWING X_i AND X_{i+1}

The slope of the secant line through points P and Q is given by:

$$[f(x_{i+1}) - f(x_i)]/[x_{i+1} - x_i]$$

Letting $dx_{i+1} = x_{i+1} - x_i$ we have as the equation of the secant line through points P and Q:

$$y - f(x_{i+1}) = ([f(x_{i+1}) - f(x_i)]/dx_{i+1})*(x - x_{i+1})$$

Hence,

(1) $$x_{i+2} = x_{i+1} + dx_{i+2}$$

where

(2) $$dx_{i+2} = \frac{(dx_{i+1})*f(x_{i+1})}{f(x_i) - f(x_{i+1})}$$

The initial value x_0 is input by the user and dx_0 is usually taken as a small fractional part of x_0

For example, if we assume $f(x_{-1})=0$ and $dx_0=(.01x_0)$ then $x_1 = (.99)x_0$

Execution halts when x_{i+2} and x_{i+1} are rounded and found to be equal.

Routine Listing For: SV

```
 91♦LBL C            108 ST* 09
 92♦LBL "SV"         109 ST- 08
 93 STO 07           110 RCL 09
 94 E                111 RCL 08
 95 %                112 X≠0?
 96 RCL Z            113 /
 97 X=0?             114 STO 09
 98 X<>Y             115 X<> 07
 99 STO 09           116 ST+ 07
100 CLST             117 RND
101♦LBL 04           118 RCL 07
102 RCL Z            119 RND
103 STO 08           120 X≠Y?
104 RCL 07           121 GTO 04
105 FS? 10           122 RCL 07
106 VIEW X           123 RTN
107 XEQ IND 06
```

LINE BY LINE ANALYSIS OF SV

Lines 91-101 initialize the program by storing the initial guess X_0 in R07 and store the initial step size in R09. Note that if the user has input 0 as the initial step size then lines 94 & 95 and lines 97 & 98 calculate and select the value $0.01*X_0$ as the actual step size.

Lines 101-121 are the main loop in the program. At LBL 04 X, Y, and T are assumed to be scratch and $f(X_i)$ is assumed to be in Z. The next approximation is calculated via formula (1) and is stored in R07 (line 116). Next, the two most recent approximations are rounded and tested for equality in line 120. A branch is then made back to LBL 04 unless the rounded values are equal.

Lines 122 & 123 recall the final solution and end the routine.

REFERENCES FOR SV

1. Forman S. Action, NUMERICAL METHODS THAT WORK, Harper and Row, New York, 1970
2. S. D. Conte and C. de Boor, ELEMENTARY NUMERICAL ANALYSIS, McGraw-Hill, 1972
3. John Kennedy (918) PPC JOURNAL "Method of Successive Bisections" V5N8P19. See also V6N5P10
4. Chris Stevens, PPC JOURNAL, V5N8P45, Sept.-Oct. 1978
5. William M. Kahan, "Personal Calculator Has Key To Solve any Equation f(x)=0", Hewlett-Packard Journal, December 1979.

CONTRIBUTORS HISTORY FOR SV

John Kennedy (918) wrote the SV program for the HP-41C from a previous HP-25 program. Graeme Dennes (1757) and Richard Schwartz (2289) made suggestions for improvements in the accuracy and overall program operation. Harry Bertucelli (3994) suggested register usage to allow SV to be used with IG. Graeme Dennes(1757) and Iram Weinstein (6051) contributed to the documentation of SV.

FINAL REMARKS FOR SV

SV needs improvement in almost all areas. SV is only a basic routine designed to be used primarily from the keyboard where the user may watch the convergence (or lack thereof) and take action to halt SV and make a new guess. SV lacks all of the sophistication of the SOLVE function on the HP-34C calculator.

FURTHER ASSISTANCE ON SV

John Kennedy (918) phone: (213) 472-3110 evenings

Graeme Dennes (1757) phone (415) 592-2957 evenings

NOTES

TECHNICAL DETAILS

XROM: 20, 10 — **SV** — SIZE: 010 minimum

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
- R00: not used
- R06: function LBL name
- R07: point x_i
- R08: $f(x_i)$
- R09: dx_i
- R10: not used
- R11: not used
- R12: not used

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: displays successive approximations when F10 is set
- 25: not used

Display Mode: SCI n recommended controls accuracy

Angular Mode: not used, but may be required by function

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
function LBL in R06	none

Local Labels In This Routine: C, 04

Execution Time: variable, depends on function, display setting, and initial guess.

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: IG

Bytes In RAM: 51

Registers To Copy: 43

Other Comments:

APPENDIX L - SPECIAL CHARACTERS - SC

This routine was planned to be included in the ROM until it was replaced by the matrix routines M1 through M5, which were felt to be more useful. Special Characters extends the 82143A printer standard character set by an additional 60 characters. These include subscripts, superscripts, math and game symbols, and more frequently used greek letters. A complete list of all the symbols included appears in table 1, below.

SC - SPECIAL CHARACTERS
standard character set

X	CHARACTER SF10	CHARACTER CF10	X	CHAR.
0	o	o	29	≈
1	1	1	30	≡
2	2	2	31	≤
3	3	3	32	≥
4	4	4	33	∞
5	5	5	34	∇
6	6	6	35	Π
7	7	7	36	∈
8	8	8	37	σ
9	9	9	38	ν
10	x	x	39	Ψ
11	y	y	40	ω
12	z	z	41	⊗
13	+	+	42	™
14	-	-	43	⚀
15	/	/	44	⚁
16	(	(	45	⚂
17	)	)	46	⚃
18	▫		47	⚄
19	√		48	⚅
20	∫		49	♦
21	dx		50	♥
22	dy		51	♠
23	dt		52	♣
24	∂		53	◀
25	±		54	▶
26	⇄		55	<
27	~		56	>
28	≃			

Table 1. The complete list of new symbols added to the printer's standard ACCHR set by use of the Special Characters routine. This table was produced by the 'SCDEMO' program.

Note that the first 18 characters, numbered 0 through 17, produce superscripts if F10 is clear and subscripts if F10 is set. The remainder, characters 18 through 56, are unaffected by the status of flag 10. A listing of the program to produce table 1 is presented here:

BAR CODE ON PAGE 482

APPLICATION PROGRAM FOR: **SC**

```
01•LBL "SCDEMO"
02 "SC - SPECIAL"
03 "├ CHARACTERS"
04 ACA
05 PRBUF
06 " standard ch"
07 "├aracter set"
08 ACA
09 PRBUF
10 ADV
11 "   CHARACTER"
12 ACA
13 PRBUF
14 SF 12
15 "X"
16 ACA
17 CF 12
18 " SF10 CF10   "
19 ACA
20 SF 12
21 "X"
22 ACA
23 CF 12
24 "  CHAR."
25 ACA
26 PRBUF
27 FIX 0
28 CF 00
29 CF 29
30 CF 12
31 .009
32 STO 00
33 29
34 STO 01
35•LBL 00
36 FS? 00
37 GTO 02
38 1
39 SKPCHR
40•LBL 02
41 RCL 00
42 INT
43 CLA
44 ARCL X
45 ACA
46 3
47 SKPCHR
48 SF 12
49 RDN
50 SF 10
51 XEQ "SC"
52 ACSPEC
53 CF 12
54 3
55 SKPCHR
56 SF 12
57 RCL 00
58 CF 10
59 XEQ "SC"
60 ACSPEC
61 CF 12
62 3
63 SKPCHR
64 CLA
65 ARCL 01
66 ACA
67 3
68 SKPCHR
69 SF 12
70 RCL 01
71 XEQ "SC"
72 ACSPEC
73 PRBUF
74 CF 12
75 1
76 ST+ 01
77 ISG 00
78 GTO 00
79 FS? 00
80 GTO 03
81 10.017
82 STO 00
83 SF 00
84 GTO 00
85•LBL 03
86 18.028
87 STO 00
88•LBL 01
89 CLA
90 ARCL 00
91 ACA
92 5
93 SKPCHR
94 SF 12
95 RCL 00
96 XEQ "SC"
97 ACSPEC
98 CF 12
99 5
100 SKPCHR
101 RCL 01
102 57
103 X=Y?
104 GTO 04
105 X≠Y?
106 RDN
107 ACX
108 3
109 SKPCHR
110 SF 12
111 RDN
112 XEQ "SC"
113 ACSPEC
114•LBL 04
115 PRBUF
116 CF 12
117 1
118 ST+ 01
119 ISG 00
120 GTO 01
121 END
```

The barcode for both SC and SCDEMO appear in appendix K of this manual.

APPENDIX L CONTINUED ON PAGE 429.

SX - STORE Y IN ABSOLUTE ADDRESS X

SX can be used to store data or program bytes in any desired register in user memory. **SX** permits direct modification of programs and key assignments, or storing data in the unused memory space between the .END. and the key assignment registers (this is especially useful when page switching).

Example 1: Perform the following steps:

1. GTO..
2. Resize if necessary to get at least 4 program registers available
3. In PRGM, key in "ABCDEFGHIJ" to open up 11 bytes, then delete the line.
4. In RUN mode, key in "9F7F1C1BA2801C"

```
XEQ HN
XEQ E?
1        (Store 1 register
+          above .END.)
XEQ SX
XEQ GE.
```

5. SST in PRGM mode and you'll see that the 7 bytes you coded up using **HN** are now program instructions. Some unusual ones are included for your study.

Example 2: Continuing the above example, GTO.000 and make sure there are at least 2 unused program registers. Now key in (in RUN mode)

```
π
XEQ E?
1
-
XEQ SX
XEQ GE.
```

Line 00 now shows no program registers left, because π is sitting right below the .END.. To clean up key in

```
CLST
XEQ E?
1
-
XEQ SX
XEQ GE.
```

This sequence stores a zero (7 nulls) below the .END., restoring use of the remaining program registers. Note that π was recalled from the register below the .END., appearing in the Z register.

COMPLETE INSTRUCTIONS FOR SX

1. Place the desired code in Y and its destination (absolute decimal address) in X. X must be at least 192 and less than 256 + n*64, where n is the number of (single density) RAM modules present.

2. XEQ **SX** to store the code. Stack usage is as shown:

	Before		After
T	don't care	T	temporary c register (from **OM**)
Z	z	Z	former contents of absolute address
Y	code	Y	z
X	absolute address	X	code
L	don't care	L	absolute address -16
alpha	don't care	alpha	cleared

TECHNICAL DETAILS

XROM: 10,56 — **SX** — SIZE: 000

Stack Usage:
- 0 T: temporary c
- 1 Z: old reg. contents
- 2 Y: Z
- 3 X: Y
- 4 L: X-16

Flag Usage: SEVERAL USED BUT ALL RESTORED
04:
05:
06:
07:
08:
09:
10:
25:

Alpha Register Usage:
- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊢: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: USED BUT RESTORED
- 14 d: NOT USED
- 15 e:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 4

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Global Labels Called:

Direct	Secondary
OM	PART OF GE
PART OF RX	

Local Labels In This Routine: NONE

Execution Time: 1.1 second.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO!
Program File:	IF
Bytes In RAM:	16
Registers To Copy:	60

Other Comments: If you get NONEXISTENT, SST past line 126 and R/S to restore the curtain.

SX

3. Note that the former contents of the chosen register appear in Z. To restore these contents use X<>Z LASTx 16 + XEQ **SX**.

4. The temporary c register (with the curtain at 16_{10}) is left in T. It can be recognized by its appearance in the display as ⊠ 天 ⊠ 天 天 ⊠ , although the last two characters may be different. The second full man character (in the fourth position) conclusively identifies this as a temporary c register, rather than a normal one.

5. If you get NONEXISTENT when using **SX**, go into PRGM mode, SST once to get to line 127, go back to RUN mode and R/S. This puts the old curtain back in place.

6. For an explanation of HP-41C memory structure, see **LF** Figure 1.

LINE BY LINE ANALYSIS OF SX

Line 123 executes the curtain lowering routine **OM** and stores X-16 in Lastx. Lines 124 and 125 bring the code into X and Y, then line 126 stores it in the chosen register. Lines 132-136 restore the original curtain and bring the code back into X.

CONTRIBUTORS HISTORY FOR SX

SX substitutes for, and to some extent supersedes, Bug 2 as a way of storing a code into any register in user memory. This concept originated with Bill Wickes' (3735) B2 program (see *PPC CALCULATOR JOURNAL*, V7N3P7a). Similar routines were written by John MCGechie (3324) and others. **SX** was written by Keith Jarett (4360) around **OM**. It is now overshadowed by **LB**, but it still can be useful.

FURTHER ASSISTANCE ON SX

Call Keith Jarett (4360) at (213) 374-2583.
Call Roger Hill (4940) at (618) 656-8825

Routine Listing For:	SX
122◆LBL "SX" 123 XEQ 14 124 X<>Y 125 ENTER↑ 126 X<> IND L 127 RDN 128 GTO 13 132◆LBL 13 133 X<>Y 134 X<> c 135 RDN 136 RTN 137◆LBL 14 138 16	139 - 140 SIGN 141 RDN 142◆LBL "OM" 143 XEQ 14 144 "▮*i*" 145 X<> [146 STO \ 147 "├**" 148 X<> \ 149 CLA 150 X<> c 151 RTN

NOTES

Sb - STORE b IN ROM

Sb provides a quick way to transfer program execution to any point in any ROM. If a RTN is required XE should be used.

Ab and Sb are both one-instruction "programs" that provide an ultra-fast ROM entry capability as an alternative to XE (XROM entry). Ab consists of an ASTO b instruction which stores a user-specified code in the program pointer, immediately transferring execution to another point in ROM. XE can be thought of as an XEQ IND function. Similarly, Ab and Sb can be thought of as GTO IND functions. XE, Ab, and Sb use the contents of an indirect register (ALPHA or X) as an entry address. XE preserves up to five subroutine returns including the one to the calling RAM program. On the other hand, Ab and Sb destroy all pending subroutine returns and program execution ultimately halts in ROM, unless synthetically constructed returns were provided.

Example 1: The following routine demonstrates the use of Ab (or Sb) in a RAM program. This program has these features:

1) It is named "VK" which is the name of the PPC ROM routine which is to be enhanced/modified. The RAM "VK" rather than the ROM VK will be called by an XEQ "VK", because CATALOG 1 global labels are found before CATALOG 2 global labels or functions during a search.

2) It adds Roger Hill's (4940) original "KEYS USED:" as a replacement for the stalled "flying goose".

3) It bypasses Lines 01-07 of the PPC ROM routine VK which allows "VK" to operate with the printer present, but turned off. This routine will not work with the printer present and turned on.

USING Ab:

```
01  LBL "VK"
02  CF 21
03  "KEYS USED:"
04  AVIEW
05  hex F2 W9 33
06  XROM Ab
```

USING Sb:

```
01  LBL "VK"
02  CF 21
03  "KEYS USED:"
04  AVIEW
05  hex F2 W9 33
06  RCL M
07  XROM Sb
```

Line 05 is a two byte text line which represents the address in ROM to which the GTO IND is to jump. The rightmost three hexadecimal digits (933 in this case) give the location of the jump destination within a 4K ROM. (Note that there are 4096 possible values for three hex digits). The remaining, or leftmost, hexadecimal digit indicated by W here, gives the port address of the 4K ROM as follows: 8 = Port 1 lower 4K, 9 = Port 1 Upper 4K, A = Port 2L, B = Port 2U, C = Port 3L, D = Port 3U, E = Port 4L, F = Port 4U. Since VK is in the lower 4K of the PPC ROM (the lower 4K appears first in CATalog 2), W will be 8, A, C, or E here according as the PPC ROM is in port 1, 2, 3, or 4. Line 05 can be created using LB or the text Q-loader (see *PPC CALCULATOR JOURNAL*, V7N8P27a).

In the "VK" examples above, the use of Ab is preferred to Sb because it saves bytes. In other applications for which the NNN is already in the X register, Sb might be preferred. Both examples above transfer program execution to line 08 of PPC ROM routine VK and the program ultimately halts in the PPC ROM.

Example 2: The two byte text line which is created for the above examples makes the RAM "VK" routine port-dependent; that is, the routine will work correctly only if the PPC ROM is installed in the port represented by W. This can be remedied by the PPC ROM routine Rb. The following routine calls a RAM routine "PE" (PPC ROM ENTRY) which in turn calls PPC ROM routine Rb. Rb recalls the contents of register b which contains the address (including the port number) of that point in the PPC ROM. "PE" modifies the two-byte pointer which is provided by the calling program (the RAM "VK" routine in this case) to give it the port number of the PPC ROM which was obtained by Rb. The pointer supplied to "PE" should be either 8 ijk or 9 ijk depending upon which half of the PPC ROM is to be entered. In effect the calling RAM program addresses Port 1 and the "PE"/Rb combination modifies the call to address the correct port. Thus, the calling RAM program becomes port-independent.

USER PROGRAM:

```
01  LBL "VK"
02  CF 21
03  "KEYS USED:"
04  Hex F2 89 33
05  XEQ "PE"
06  XROM Sb
```

PPC ROM ENTRY SETUP PROGRAM:

```
01  LBL "PE"
02  XROM Rb
03  X<>M
04  Hex F6 7F 00 00 00
    00 00
05  X<>M
06  Hex F6 7F 00 00 00
    00 00
07  X<>d
08  .
09  FS? 02
10  E↑X
11  2
12  FC? 01
13  CLX
14  +
15  X<>Y
16  X<>M
17  X<>d
18  RDN
19  2
20  /
21  INT
22  X≠0?
23  SF 01
24  LASTX
25  FRC
26  X≠0?
27  SF 02
28  X<>M
29  X<>d
30  X<>M
31  Hex F3 7F 00 00
32  X<>N
33  CLA
34  END
```

COMPLETE INSTRUCTIONS FOR Sb

At first glance, Ab and Sb don't seem to offer any byte savings in the user's RAM programs. XROM Ab and XROM Sb each require two bytes of RAM which is the same as ASTO b and STO b require; therefore, the Ab and Sb labels appear to be taking up space in the PPC ROM without purpose. However, this is not the case as we shall see.

For example, with the printer connected do the following:

0) Assign RCL b and STO b to keys (use MK).
1) GTO "PRAXIS".
2) RCL b (or ARCL b).
3) XEQ CATALOG 1.
4) STO b (or ASTO b).
5) PRGM mode on (Note: program pointer is not at "PRAXIS").
6) PRGM mode off.
7) GTO "PRPLOT" (or any ROM global other than "PRAXIS").
8) STO b (or ASTO b).
9) PRGM mode on (Note: program pointer is at "PRAXIS").
10) PRGM mode off.

Why did steps 7 to 9 produce the desired result (i.e, locate "PRAXIS") while steps 3 to 5 did not?

The GTO "PRAXIS" set the program pointer (in register b) at ROM address 6108 (i.e., byte 108_{16} of ROM 6 - the printer). The RCL b brought this value (6108) to the X register. Execution of CATALOG 1 reset the program pointer to a RAM address (in a user program). When in RAM, the address 6108 is interpreted as byte 6 of register 108_{16}. Thus, the subsequent STO b sets the program pointer to this location in RAM (if it exists and is occupied) and not back to the location of "PRAXIS".

However, GTO "PRPLOT" sets the program pointer to a ROM address, so that when another STO b is executed, 6108 is once again interpreted as a ROM address and the program pointer is reset to "PRAXIS".

PPC ROM routines Ab and Sb automate the manual process given in the above example. If the ALPHA or X register contains a code which represents an address (e.g., 6108), then XROM Ab or XROM Sb will cause this address to be interpreted as a ROM address. When an user's calling program encounters XROM Ab or XROM Sb , program execution is transferred to the PPC ROM. Since the program pointer now represents a ROM address (in the PPC ROM), the subsequent ASTO b or STO b (in Ab or Sb) will cause the running program to jump to the ROM address (e.g., 6108) which the user provided in the ALPHA or X register. Program execution will continue until an END, RTN or STOP is encountered in the ROM program, at which time a halt will occur with the program pointer in ROM.

Note that in the above example, program execution jumped from a user's RAM program to the PPC ROM and then to the printer ROM. This all occurred as the result of XROM Ab or XROM Sb , two short but powerful instructions.

Ab and Sb destroy all subroutine returns that are pending at the time Ab or Sb is executed. If the pending returns are needed, then the PPC ROM routine XE should be used instead of Ab or Sb .

LINE BY LINE ANALYSIS OF Sb

See the Complete Instructions above for the theory of the operation of Sb.

CONTRIBUTORS HISTORY FOR Sb

Ab and Sb owe their existence to Tom Cadwallader (3502). Tom requested their inclusion after Bill Pickard (3514) discovered that STO b behaved differently when the program pointer was already in ROM. Bill had been trying to get into internal ROM 0 using Charles Close's (3878) "ROM + " program (see *PPC CALCULATOR JOURNAL*, V8N1P14). The STO b behavior described here was also discovered independently by Robert Groom (5127). The "PE" application routine was written by Tom Cadwallader (3502).

FINAL REMARKS FOR Sb

The HP-41C's MPU apparently has some means (Flag ?) of knowing whether the ROM instructions that it is executing are user language or assembly language. When we learn how to make the MPU recognize that ROM contents at a given point are assembly language, we can then use Ab and Sb to begin execution at that point in an assembly language interpretation.

TECHNICAL DETAILS

XROM: 20,01 — **Sb** — SIZE: 000

Stack Usage:
0 T:
1 Z: ALL UNCHANGED
2 Y:
3 X:
4 L:

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Alpha Register Usage:
5 M:
6 N: ALL UNCHANGED
7 O:
8 P:

Other Status Registers:
9 Q: NOT USED
10 ⊦: NOT USED
11 a: NOT USED
12 b: USED
13 c: NOT USED
14 d: NOT USED
15 e:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 0-6, DEPENDING ON THE CODE STORED.

ΣREG: UNCHANGED

Data Registers: NONE USED

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: LESS THAN .1 second.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	NO
Program File:	SR
Bytes In RAM:	8
Registers To Copy:	40

Other Comments:
Stored program pointer will be interpreted as a ROM pointer.

Sb won't work in RAM.

Routine Listing For: Sb	
24♦LBL "Sb" 25 STO b	

FURTHER ASSISTANCE ON

Call Tom Cadwallader (3502) at (406) 727-6869.
Call Roger Hill (4940) at (618) 656-8825.

T1 - BEEP ALTERNATIVE

T1 is an unusual sequence of thirteen TONE 57's and TONE 89's that may be used as a BEEP alternative. T1 may be executed at any time, although Flag 26 should be set to hear the tone.

Example 1: An alarm is desired to indicate the completion of a program. The alarm should sound continuously until stopped.

```
01  LBL 01        100 loops were
02  XROM T1       timed at 62.8
03  GTO 01        seconds.
```

Example 2: The alarm of example 1. is too fast and is to be slowed down. The routine is modified by removing the PPC ROM and keying XEQ T1 in place of line 02 and plugging in the PPC ROM again. The extra time for RAM search slows the loop time, so that 100 loops now take 104.2 seconds. This simple technique works, but is not practical even if the time is as required. Do you see why? Hint: Byte count.

COMPLETE INSTRUCTIONS FOR T1

T1 requires no inputs and may be executed any time a 0.6 second TONE burst is desired.

Routine Listing For: T1	
140♦LBL "T1"	148 TONE 9
141 TONE 7	149 TONE 9
142 TONE 7	150 TONE 7
143 TONE 7	151 TONE 9
144 TONE 9	152 TONE 9
145 TONE 9	153 TONE 9
146 TONE 9	154 RTN
147 TONE 7	

LINE BY LINE ANALYSIS OF T1

The T1 routine consists of 13 synthetic tones running from line 141 thru line 153. The first TONE is TONE 57 and displays as TONE 7. The other tone is TONE 89 which displays as TONE 9. The complete sequence takes 0.6 seconds.

REFERENCES FOR

See *PPC CALCULATOR JOURNAL*, V7N10P8c.

CONTRIBUTORS HISTORY FOR T1

Cary Reinstein (2046) developed T1 for his 4 TREK program. The tone sequence was used as a phaser sound in that STARTREK like game.

FINAL REMARKS FOR T1

T1 is a representative example of synthetic tone sequences that are effective in programs. TONE instructions take two bytes each and unique sounds may be created with many, many TONES. ROM is an effective means to implement these sounds

FURTHER ASSISTANCE ON T1

Richard Nelson (1) -- (714) 754-6226 P.M.
Richard Schwartz (2289) -- (213) 447-6574 Evenings.

TECHNICAL DETAILS

XROM: 10,47 | T1 | SIZE: 000

Stack Usage:
0 T:
1 Z:
2 Y:
3 X:
4 L:
STACK NOT USED

Alpha Register Usage:
5 M:
6 N:
7 O:
8 P:
NONE USED BY ROUTINE

Other Status Registers:
9 Q:
10 ⊢:
11 a:
12 b:
13 c:
14 d:
15 e:
NONE USED BY ROUTINE

ΣREG: NOT USED

Data Registers:
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:
NONE USED BY ROUTINE

Flag Usage: NONE
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: N/A

Angular Mode: N/A

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: < 0.7 seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? YES
Program File: BL
Bytes In RAM: 35
Registers To Copy: 46

Other Comments:

T1

APPENDIX L CONTINUED FROM PAGE 423.

COMPLETE INSTRUCTIONS FOR SC

Since this is a RAM program, one must first load it into the 41C, either by scanning the barcode or reading magnetic cards recorded earlier. Now, each time a character from the set in SC is desired, one needs only to place the character number into X and XEQ SC. The synthetic text string corresponding to the special printer character will be placed in X, to either be placed immediately into the print buffer by ACSPEC, or stored in a data register for later use.

The first 18 characters may be printed as either superscripts (by clearing flag 10) or subscripts (by setting flag 10). If flag 10 is set while accessing a character which has only one form (characters #18 - #56), the character will not be printed correctly. Therefore F10 should remain clear during the use of these characters. After F10 is set before executing SC, the flag is automatically cleared so no characters are accidentally modified.

One efficient use of SC would be to load all the desired characters into data registers first, and then to recall them when needed. An example of this would be if a program using the symbols of the six faces of dice. Once the text strings are in six registers, they are later recalled and ACSPEC'ed into the print buffer. Thus the SC program would only have to be called once for each different character desired, rather than each time the character was required.

A convenient routine for exploring the special characters is labeled PSC below:

```
01  LBL PSC        04  STOP
02  XEQ  SC        05  PRBUF
03  ACSPEC         06  RTN
```

Read the SC program into the HP-41. Key the routine and assign LBL PSC to a key. ENTER the number of the symbol and press the PSC key. If you want it to print, press R/S. Build up the buffer with up to six SC symbols (no spaces) using the 82143A printer.

Example 1. Print the following lines on the printer using the SC program:

H2O ⟷ H+ + OH-
∫eX dx = eX
-b + SQR (b2-4ac)/2a
(Print the 4 phases of the moon)

```
01♦LBL "H2O"        14 "H"
02 SF 12            15 ACA
03 "H"              16 CF 10
04 ACA              17 13
05 SF 10            18 XEQ "SC"
06 2                19 ACSPEC
07 XEQ "SC"         20 "+OH"
08 ACSPEC           21 ACA
09 "O"              22 14
10 ACA              23 XEQ "SC"
11 26               24 ACSPEC
12 XEQ "SC"         25 PRBUF
13 ACSPEC           26 END
```

$H_2O \leftrightarrow H^+ + OH^-$

```
01♦LBL "eX"         13 SKPCOL
02 SF 12            14 21
03 20               15 XEQ "SC"
04 XEQ "SC"         16 ACSPEC
05 ACSPEC           17 " = e"
06 "e"              18 ACA
07 ACA              19 10
08 CF 10            20 XEQ "SC"
09 10               21 ACSPEC
10 XEQ "SC"         22 PRBUF
11 ACSPEC           23 END
12 2
```

$\int e^x dx = e^x$

```
01♦LBL "QU"         12 ACSPEC
02 CF 12            13 "(b"
03 "-b "            14 ACA
04 ACA              15 CF 10
05 25               16 2
06 XEQ "SC"         17 XEQ "SC"
07 ACSPEC           18 ACSPEC
08 " "              19 "-4ac)/2a"
09 ACA              20 ACA
10 19               21 PRBUF
11 XEQ "SC"         22 END
```

$-b \pm \sqrt{}(b^2-4ac)/2a$

```
01♦LBL "PH"         14 ACSPEC
02 CF 12            15 1
03 "MOON PHASES:"   16 SKPCOL
04 PRA              17 RDN
05 53               18 ACSPEC
06 XEQ "SC"         19 55
07 ACSPEC           20 XEQ "SC"
08 1                21 ACSPEC
09 SKPCOL           22 56
10 RDN              23 XEQ "SC"
11 ACSPEC           24 ACSPEC
12 54               25 PRBUF
13 XEQ "SC"         26 END
```

MOON PHASES:
◖●◗○

FURTHER DISCUSSION OF SC

For those who do not wish to load the entire 500-plus bytes of SC into RAM memory each time a handful of special characters is desired, the barcodes below will suffice. These are the data barcodes for the individual characters, which can be scanned directly into the ALPHA register or into a program line. WARNING: Many of these codes will not operate correctly if scanned in normal mode. Those containing bytes from row zero of the hex table will usually lock up the calculator when scanned if not in program mode. They do operate correctly, however, as program lines.

The codes, below, which lock up the 41C if scanned in normal mode are marked with an asterisk.

APPENDIX L CONTINUED ON PAGE 439.

TB - BASE TEN TO BASE B

This base conversion routine is from base 10 to base b where b lies in the range 2<=b<=19. The base b is stored in a data register and the final result is returned in the alpha register. TB takes its base 10 input from the X-register. Setting a flag will cause the contents of the alpha register to be displayed. The capabilities of the alpha register character display are needed when b>10. See also the routine BD. This routine is the inverse of BD.

Example 1: Convert 12773 to base 12.

Store the base 12 in R06. 12 STO 06. Set flag 10 to automatically display the result from the alpha register. SF 10. Key 12773 and XEQ "TB". The result displayed is 7485'. The single quote character following the final digit is a reminder that the number displayed is not a base 10 number.

Example 2: Convert 20284 to base 16.

Store 16 in R06. 16 STO 06. Leave flag 10 set from the previous example. Key 20284 and XEQ "TB". The result displayed is 4F3C'.

COMPLETE INSTRUCTIONS FOR TB

1) To convert a base 10 integer to its representation in base b first store the base b in R06.

2) An option is to display the final result by setting flag 10. If flag 10 is clear the final result will remain in the alpha register and will not be displayed when the routine ends.

3) Key in the base 10 number in X and XEQ "TB".

4) If flag 10 is set the display will show the base b representation with a single quote following the rightmost digit. This indicator is simply a reminder that the number displayed is not a base 10 number. If flag 10 is clear the final answer is in the alpha register. The output is limited to 14 digits including the final single quote. It is possible to overflow the display by choosing a combination of a small enough base b and a large base 10 number.

The stack input/output for TB is as follows:

Input:	Output:
T: T	T: 0
Z: Z	Z: 0
Y: Y	Y: 0
X: base 10 number	X: 0
L: L	L: *
M: M	M: } base b result
N: N	N: } base b result
O: O	O: clear

MORE EXAMPLES OF TB

Example 3: Some versions of BASIC and Pascal support what is called a long integer which is a 32-bit number. If the maximum value is 2,147,483,647 find its hexadecimal base 16 representation.

First store 16 in R06. Set flag 10. Then key in 2,147,483,647 and XEQ "TB". See 7FFFFFFF' returned.

Example 4: The largest integer the HP-41C will hold is 9999999999. Find the hex equivalent of this number.

If 16 remains stored in R06 from the previous example simply key in 9999999999 and XEQ "TB". See 2540BE3FF'.

FURTHER DISCUSSION OF TB

The TB routine is not the fastest possible routine for doing base conversions on the 41C but other methods were not used because they are very base dependent. For fast decimal to base b conversions on the HP-41C study the technique used in the routine by John Kennedy (918) which appeared in PPCJ V7N4P22. TB was chosen because it does not require the use of other data registers and it can be applied to a wide variety of bases.

Routine Listing For: TB

```
36♦LBL B
37♦LBL "TB"
38 "'             "
39 RCL [
40 X<>Y

41♦LBL 03
42 ENTER↑
43 INT
44 RCL 06
45 MOD
46 9
47 -
48 X>0?
49 ISG X

50♦LBL 04
51 39
52 +
53 10↑X
54 STO ]
55 "├       "
56 CLX
57 X<> ]
58 X<> \
59 STO [
60 RDN
61 RCL 06
62 /
63 INT
64 X≠0?
65 GTO 03

66♦LBL 05
67 "├ "
68 CLX
69 RCL [
70 X≠Y?
71 GTO 05
72 CLX
73 X<> ]
74 X<> \
75 STO [
76 CLST
77 FS? 10
78 XROM "VA"
79 RTN
```

LINE BY LINE ANALYSIS OF TB

Lines 36-40 initialize the routine. Line 36 automatically assigns TB to key B when the program pointer is stopped in this section of ROM. Line 38 is a synthetic text line which stores a single quote followed by 13 blanks. Lines 39-40 recall 7 blanks into the stack. These blanks float up and down in the stack throughout the main loop in the routine and are not used until line 70.

Lines 41-65 are the main loop in the routine. At the start of LBL 03 the remaining base 10 number is in X and the 7 blanks are in Y. The base b digits are built up starting with the least significant digits. The base 10 equivalent is computed at lines 44 and 45. Lines 46-53 convert this decimal number to its appropriate alpha equivalent where it is stored in the O register (line 54). Line 54 acts as an append to the remaining characters in alpha and lines 55-59 serve as an alpha shift function to prepare M and N for the next character (digit) to be appended. Lines 60-63 calculate the remaining base 10 result and a branch is made back to LBL 03 as long as this number is nonzero.

Lines 66-71 pad blanks in alpha so the result is left-justified. At line 70 the 7 blanks are used in a comparison test.

Lines 72-75 serve as an alpha shift so the final digits are in M and N.

Lines 76-79 finish the routine by clearing the stack and performing the ROM "alpha view" function if flag F10 was set.

REFERENCES FOR TB

1. HP-25 Library, 65 NOTES V4N4P8b.
2. George Eldridge (5575), PPC Calculator Journal, "HP-41C HEX TO/FROM DECIMAL" V7N9P31B.
3. John Kennedy (918), PPC Calculator Journal, "Decimal to Hex Routine" V7N4P22
4. Other earlier PPC related matter may be found in: V4N3P14d V5N2P12b V5N3P21b

CONTRIBUTORS HISTORY FOR TB

George Eldridge (5575) wrote the TB routine. John Kennedy (918) wrote the documentation for TB .

FURTHER ASSISTANCE ON TB

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

TECHNICAL DETAILS

TB — XROM: 20, 18 — SIZE: 007 minimum

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: used to
- 6 N: accumulate base b result
- 7 O: used
- 8 P: used

Other Status Registers:
- 9 Q: not used
- 10 ⊦: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
- R00: not used
- R06: base b
- R07: not used
- R08: not used
- R09: not used
- R10: not used
- R11: not used
- R12: not used

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: To display result in alpha reg., set F10
- 25: used in VA routine

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
VA if F10 set	none

Local Labels In This Routine: B, 03, 04, 05

Execution Time: 1.8 seconds minimum, plus approximately 0.87 seconds per digit (character) in alpha

Peripherals Required: none

Interruptible? yes

Execute Anytime? none

Program File: BD

Bytes In RAM: 90

Registers To Copy: 53

Other Comments:

A missing final quote is a sure indication of overflow. Output is limited to 13 digits plus the final quote.

TN - TONE N (0-127)

TN is a demonstration/synthetic programming example routine that converts a TONE number in the X register into a synthetic tone instruction and executes it. **TN** is slow and is not normally used for synthetic tones in a program. **MK**, or **LB** are more suitable to place synthetic tones into your programs. **TN** can be regarded as a TONE IND X instruction where X may be any number from 0 to 127. WARNING: DO NOT PRESS R/S AFTER EXECUTING **TN** FROM THE KEYBOARD. When **TN** is executed from the keyboard, the routine stops in ROM. Pressing R/S causes execution of the following routine, **CX** which is almost certain to cause MEMORY LOST.

Example 1: Paz wants to demonstrate the full range of tones to a friend. She assigns **TN** to the CHS Key and keys various tone numbers pressing the assigned key after each.

26 **TN** Longest duration and lowest frequency TONE
106 **TN** Shortest duration of lowest frequency TONE
37 **TN** Shortest duration TONE
57 **TN** Shortest duration of highest frequency TONE
25 **TN** Longest duration of highest frequency TONE

Example 2: "Play" all 128 synthetic tones of the HP-41 using a short routine.

```
01  FIX 0          06  XROM TN
02  .127           07  ISG X
03  LBL 01         08  GTO 01
04  VIEWX          09  END
05  ENTER↑
```

The TONE Number is displayed, followed by the actual TONE.

COMPLETE INSTRUCTIONS FOR TN

TN requires a valid numeric input for proper operation. Once the routine is finished, however, a non-normalized number (-0, HEX 80 00 00 00 00 00 00) is left in Z. Negative numbers also work, but the -0 won't work and will stop showing "ALPHA DATA" in the display. If **TN** is executed three times with a cleared stack, the -0 will propagate to fill X, Y, and Z. The fourth execution will then stop showing "ALPHA DATA". Once a TONE is heard, you may press LASTX followed by **TN** for a repeat.

MORE EXAMPLES OF TN

Example 3: A short "chirp" is desired every two seconds. The loop that produces the chirp is:

```
01  LBL 01
02  89
03  XROM TN
04  GTO 01
```

Example 4: An infrequent random tone is desired using TONES 0 to 127. The mean of 63 and standard deviation of 21 is to be used. A routine to generate random tones under these conditions would be:

```
01  LBL 02         08  STO 00
02  63             09  LBL 03
03  STO 06         10  0
04  21             11  XROM GN
05  STO 07         12  XROM TN
06  PI             13  GTO 02
07  1/X
```

A seed of 1/π is stored in R00 and is used in line 10. The Gaussian Random Number Generator routine is used to generate the TONE numbers.

FURTHER DISCUSSION OF TN

TN demonstrates in a simple way the synthetic tones that are possible on the HP-41C/CV. Since the tone frequency and duration use data from the internal ROM's, it may be possible to identify a given ROM revision by executing a tone that has been observed to sound differently on various 41's tested.

HP-41's having the ROM revisions shown in the table were tested by executing TONE Z (TONE 113). The TONE routine is known to be in ROM 1 and TONE Z may be used to identify a given ROM revision. When more users are able to conveniently produce all HP-41 TONES a more completed table could be assembled.

TONE Z

HP-41 ROM REVISION:	Frequency	Duration
0:D 1:D 2:D	1	0.62
0:D 1:E 2:E	1	?
0:D 1:F 2:F	1	0.064

If your TONE Z is very short, you have an "F" revision (the latest) of ROM 1.

Synthetic tones have been used for the following.

- Indicator of program status/progress
- Input prompt
- Output identification
- Entertainment (music?)
- Morse Code learning and practice
- Control of other equipment e.g.:
  - a. Telephone dialers
  - b. Photo enlarger timer
  - c. Slide projector controller (1-5 channels)
- Sound effects (e.g., ticking of clock)

TN provides a convenient means to demonstrate and test synthetic tones on the HP-41C/CV. If you experiment with any of the 16 different frequency tones, you may want to identify the frequency and duration of an unknown tone. A convenient, if only, way to do this, is to place a sequence of tones in memory with the unknown tone mixed with close known tones. When you have identified the correct frequency, verification is easily made by alternating the unknown with a "known" frequency of the closest duration. Use the TONE table below.

APPLICATION PROGRAM 1 FOR TN

Sharyle wants to impress her magician husband, Barry, with her "magic" PPC ROM. She decides that an audio demonstration would be appropriate, so she wrote a program using **TN** that would produce and display all TONES from TONE 0 up to the limit of the HP-41. Sharyle's program is shown below. Synthetic TONES are valid for arguments of 0 thru 127. Inputs of

128 thru 255 will produce indirect TONES. If the register is NONEXISTENT, no sound will be heard. This may be demonstrated by the TN DEMO program and watching the tone numbers. You will have to test larger numbers. If the program pointer is in the TN DEMO program, the TONE number sequence may be started with any value by keying start TONE number, ENTER, XEQ 01. The program does not have a stop test and will run until -0 or other invalid input causes it to stop.

APPLICATION PROGRAM FOR:	TN
01♦LBL "TN DEMO"	08 AVIEW
02 FIX 0	09 XROM "TN"
03 CF 29	10 1
04 CLST	11 +
05♦LBL 01	12 ENTER↑
06 " TONE "	13 GTO 01
07 ARCL X	14 END

APPLICATION PROGRAM 2 FOR TN

The wide range of TONES available on the HP-41 provides the basis for creative programmers to use them in programs. The casual observer will be tempted to think, 'they are cute, but they have no practical value'.

Assign TN to TAN key. In PRGM key the pair 37, XROM TN six times and run these 12 program lines. Doesn't TONE 37 sound like the ticking of a clock? John McGechie (3324) utilizes tone 37 effectively in "The Charming Chiming Tick-Tocking Clock" listed in the box and described line by line below. The most convenient way to enter TONE 37 is to key 159↑ 37↑ 11 XEQ 1K, and press the A key in USER mode to enter the tone as required.

INSTRUCTIONS:

(1) Assign T to TAN, (2) key in time as HH.MMSS press TAN (3) Stop and start by R/S, rolling stack, if necessary, to recover time. (4) To use as count-down timer, CHS before XEQ. Insert a TONE or BEEP after an X = 0? conditional if an audible warning is wanted - similarly, start from zero as stopwatch. To slow, insert neutral, one byte instructions in the loop, deleting them for least slowing.

THE CHARMING CHIMING TICK-TOCKING CLOCK

01 Entry label with set time in X
02 Compensates for line 05
03 For HH.MMSS. FIX 2 for HH.MM - with 10 second rounding error
04 Seconds loop label
05 Clear ¼ hour test data
06 Second increment constant
07 Tick.Effectively TONE 37, or 25_{16}
08 Seconds increment
09 Recall alarm time
10 Alarm time?
11 Sound alarm — Optional - slows rate
12 Clear alarm time
13 1 o'clock test constant - 25 for 24 hour clock
14 Constant to Y for hour change in stack
15
16 Synthesis for X = > Y?
17 New constant for 1 o'clock
18 Doubling current time for ¼ hour and hour tests, chiming and striking
19 For constant viewing.
20 For hour and quarters test
21
22 If on the hour, exit to striking loop, label 02
23 TONE 37, half second tick
24 ¼ hour division constant
25 Determine no. of ¼ hours past the hour
26
27 Is it exactly on the ¼ hour?
28 If not, return to seconds loop start
29 Recover number of quarters past the hour
30 Return label for ¼ hour chiming loop
31 Sound chime
32 Decrement number of ¼ hours, chiming once for each quarter
33
34 Return to seconds loop after chime complete
35 Chime routine label
36 Seconds increment for chime loop time
37 Increment current time
38
39 Chime sequence
40
41
42 Clear seconds increment constant
43 Return from chime loop
44 Striking, entry label
45 Recover the current hour
46
47 Chime twice
48 Striking loop reentry label
49 Seconds increment constant for strike loop
50 Increment current time, now in Z
51 Clear seconds increment constant
52 Strike the hour
53 Decrement the hour, held in X
54 Loop back until striking complete
55 Return to seconds loop on completion of striking the hour
56 END

APPLICATION PROGRAM FOR:	TN
01♦LBL "T"	29 X<> L
02 ENTER↑	30♦LBL 04
03 FIX 4	31 XEQ 05
04♦LBL C	32 DSE X
05 RDN	33 GTO 04
06 .0001	34 GTO C
07 TONE 7	35♦LBL 05
08 HMS+	36 .0001
09 RCL 00	37 ST+ Z
10 X=Y?	38 TONE 8
11 GTO "AA"	39 TONE 7
12 RDN	40 TONE 8
13 13	41 TONE 6
14 X<>Y	42 RDN
15 X≠Y?	43 RTN
16 X>Y?	44♦LBL 02
17 1	45 X<> L
18 ENTER↑	46 XEQ 05
19 VIEW X	47 XEQ 05
20 FRC	48♦LBL 00
21 X=0?	49 .0001
22 GTO 02	50 ST+ Z
23 TONE 7	51 RDN
24 .15	52 TONE 3
25 /	53 DSE X
26 FRC	54 GTO 00
27 X≠0?	55 GTO C
28 GTO C	56 END

APPLICATION PROGRAM 3 FOR TN

TOM is experimenting with a voice recognition program on his HP-85. The program and interfacing hardware is really an amplitude/time waveform recognition system that is "taught" specific sound patterns. After studying the synthetic tones on the HP-41, Tom wonders if he could have the HP-41 "talk" to the HP-85. Looking over the HP-41 TONE table, Tom selected a three tone system using the short duration tones, TONE 70, TONE 87, and TONE 89. Three tones in combinations of three provide 3^3=27 different codes. This is adequate for the 26 letters of the alphabet. Using this concept, Tom, wrote the ALFA TN program shown below. Each letter routine is assigned to its corresponding key for demonstration and test purposes. A full alphabet sequence is accomplished by calling all 26 routines one after another. This is done under Label "=" at line 132.

Tom used a voice input TIC-TAC-TOE game on the HP-85 to test the concept. He used a bender coupler-amplifier speaker on the HP-41 and executed the sequence of ten codes (A-J) as digit inputs to the HP-85. Much to everyones amazement, it actually worked. Perhaps those tones have some use after all.

BAR CODE ON PAGE 485

APPLICATION PROGRAM FOR: TN

```
01♦LBL "ALFA TN"
02♦LBL A
03 TONE 9
04 TONE 9
05 TONE 0
06 RTN
07♦LBL B
08 TONE 9
09 TONE 9
10 TONE 7
11 RTN
12♦LBL C
13 TONE 9
14 TONE 0
15 TONE 9
16 RTN
17♦LBL D
18 TONE 9
19 TONE 0
20 TONE 0
21 RTN
22♦LBL E
23 TONE 9
24 TONE 0
25 TONE 7
26 RTN
27♦LBL F
28 TONE 9
29 TONE 7
30 TONE 9
31 RTN
32♦LBL G
33 TONE 9
34 TONE 7
35 TONE 0
36 RTN
37♦LBL H
38 TONE 9
39 TONE 7
40 TONE 7
41 RTN
42♦LBL I
43 TONE 0
44 TONE 9
45 TONE 9
46 RTN
47♦LBL J
48 TONE 0
49 TONE 9
50 TONE 0
51 RTN
52♦LBL "K"
53 TONE 0
54 TONE 9
55 TONE 7
56 RTN
57♦LBL "L"
58 TONE 0
59 TONE 0
60 TONE 9
61 RTN
62♦LBL "M"
63 TONE 0
64 TONE 0
65 TONE 0
66 RTN
67♦LBL "N"
68 TONE 0
69 TONE 0
70 TONE 7
71 RTN
72♦LBL "O"
73 TONE 0
74 TONE 7
75 TONE 9
76 RTN
77♦LBL "P"
78 TONE 0
79 TONE 7
80 TONE 0
81 RTN
82♦LBL "Q"
83 TONE 0
84 TONE 7
85 TONE 7
86 RTN
87♦LBL "R"
88 TONE 7
89 TONE 9
90 TONE 9
91 RTN
92♦LBL "S"
93 TONE 7
94 TONE 9
95 TONE 0
96 RTN
97♦LBL "T"
98 TONE 7
99 TONE 9
100 TONE 7
101 RTN
102♦LBL "U"
103 TONE 7
104 TONE 0
105 TONE 9
106 RTN
107♦LBL "V"
108 TONE 7
109 TONE 0
110 TONE 0
111 RTN
112♦LBL "W"
113 TONE 7
114 TONE 0
115 TONE 7
116 RTN
117♦LBL "X"
118 TONE 7
119 TONE 7
120 TONE 9
121 RTN
122♦LBL "Y"
123 TONE 7
124 TONE 7
125 TONE 0
126 RTN
127♦LBL "Z"
128 TONE 7
129 TONE 7
130 TONE 7
131 RTN
132♦LBL "="
133 XEQ A
134 XEQ B
135 XEQ C
136 XEQ D
137 XEQ E
138 XEQ F
139 XEQ G
140 XEQ H
141 XEQ I
142 XEQ J
143 XEQ "K"
144 XEQ "L"
145 XEQ "M"
146 XEQ "N"
147 XEQ "O"
148 XEQ "P"
149 XEQ "Q"
150 XEQ "R"
151 XEQ "S"
152 XEQ "T"
153 XEQ "U"
154 XEQ "V"
155 XEQ "W"
156 XEQ "X"
157 XEQ "Y"
158 XEQ "Z"
159 STOP
160 END
```

APPLICATION PROGRAM 4 FOR TN

This program is used with a bender coupler and tone detector that "outpulses" a relay on the telephone line for dialing purposes. The operating philosophy of the program is to prompt for a NAME? of six characters. Once a name is input, R/S causes the program to "look up" the seven digit telephone number and produce a short tone sequence for each digit. Five produces five short tones, Nine produces nine tones, Zero ten tones, etc. The "fall through" label scheme used allows a fast "pulse". This is too fast for most local offices, but is easily slowed down.

Label "DIAL" is assigned to the "D" key. ENTER is assigned to the "C" key. To dial press "D". Key NAME? after prompt, then R/S. If a new number is to be added, press "C", followed by PRGM. The Line "25 LBL:NAME?" serves as a prompt to key in a new number in the format shown below.

```
LBL ABCDEF        (up to 6 characters)
.NNNNNNN          (7 digits, could be up to
GTO 11                      10, see line 11)
```

A-F is the Alpha name, and .NN is the seven digit telephone number entered as a decimal. The GTO 11 instruction actually does the "dialing". Two telephone numbers are in the program for demonstration purposes. They may be deleted. The ? entry may be used to time a particular HP-41 for dialing speed.

Here is a line by line description of the program.

The label at line 01 provides a display description of what the key does. Line 02 is a local label used to save bytes, because it is addressed twice--lines 22 and 76. The CLX at line 03 insures that the SIN at line 06 operates on zero. Lines 04 and 05 display "AUTO DIALER" briefly while the SIN of zero is calculated. Lines 03 and 06 simply provide a delay. The NAME? prompt is displayed in lines 07 thru 09. The NAME is assumed in ALPHA when R/S causes program resumption at line 10 where ALPHA is turned off. The ISG value 0.006 is stored in R00 and the entered name is stored in the X register in lines 11 thru 13. The display shows SEARCHING using lines 14 and 15. ROM routine VA is used at lines 05, 15, 19, and 34 as good practice, even though the printer is not to be connected when using this program. Flag 25 is set at line 16 in case the indirect GTO at line 17 can't be executed. If a nonexistent label is searched for, line 17 is "skipped" and a "CAN'T FIND" display is shown by lines 18 and 19. A two second low frequency tone (TONE 30) at line 20 provides a notice of failure to find the name and a fixed duration of the CAN'T FIND display. The GTO 13 at line 21 restarts the program.

If the global label is found at line 17, the routine format is to enter the telephone number into X and go to LBL 11 at line 31. The clear flag 25 instruction at line 32 is included as good practice to avoid too wide a window of a non-indicating error condition. Lines 33 and 34 are provided to implement the philosophy that the user be kept informed as to what the program is doing. The ENTER at line 35 establishes the initial conditions required for the "dial" loop of LBL 12 at line 36. This loop consists of lines 36 thru 75 and is traversed seven times as controlled by the ISG counter value established in line 11. Lines 37 thru 42 selects the first digit to the right of the decimal point. The fractional part of the number (in both X and Y by the ENTER of line 41) is taken for the next pass of the loop. The actual series of tones is determined by line 43. If the integer part of the number in Y is two, the unconditional branch is to label 02 which is above two TONE 89's. The series of TONE 89's in lines 44 thru 73 are mixed with labels to permit entry at the appropriate point. The XROM PO's that follow each TONE provide a delay to slow down the spacing of the tones. These may be removed and replace with pairs of X<>Y or other neutral instruction as required by the telephone system or tone detect circuit. Register zero serves as a counter for seven passes through LBL 12 at line 74. When the count reaches 7, the GTO 13 at line 76 is executed and the program starts over.

The program provides a convenient means for loading new telephone numbers with lines 22 thru 24. The global label ENTER is for key assignment purposes. The RTN at line 23 stops the program pointer. When PRGM is pressed, the display shows line 24 which provides instructions for keying in the new telephone number. The "bottom-up" linkage of HP-41 global label search, will provide the shortest search time for the first entries.

The two numbers in the program illustrate the form and format of numbers in the "directory" in lines 25 thru 30. The first one is named "?". The "?" key is next to R/S and was selected for convenience of use. The "zero" telephone number at line 26 is treated as a telephone number of seven zeros. This is handy to demo the concept--no other electronics needed--and provide a means to time and test the interfacing equipment. The second telephone number is for a married couple.

This program has the advantage of minimum memory used for the number, reasonable speed, and simplified operation with only program cards to handle. An HP-41C with Quad Memory Module has a 2,237 byte capacity. Subtract 183 bytes for the basic program and 2,054 bytes remain for telephone numbers. Allow an average of 5 characters for the label (9 bytes), 8 bytes for the telephone number, and two bytes for the GTO 11 for a total of 19 bytes per telephone number entry. The number of telephone numbers an HP-41 could hold would be 108. Using the PS routine, several directories could be switched on or off as desired with the PPC ROM and QUAD switching. Thus the HP-41 becomes a truly convenient personal data base.

The program for dialing is included here primarily for ideas, because very few users will build the hardware. The program could be easily modified to simply "look up" the number and display it. Recording the program PRIVATE will keep the casual user of your machine from seeing confidential telephone numbers. A modified, display only program that used one QUAD module and holds 75 to 85 telephone numbers could be placed into the machine at the press of a switch when needed.

BAR CODE ON PAGE 485

APPLICATION PROGRAM FOR: **TN**

```
01♦LBL "DIAL"
02♦LBL 13
03 CLX
04 "AUTO DIALER"
05 XROM "VA"
06 SIN
07 "   NAME?"
08 AON
09 PROMPT
10 AOFF
11 .006
12 STO 00
13 ASTO X
14 " SEARCHING"
15 XROM "VA"
16 SF 25
17 GTO IND X
18 " CAN'T FIND"
19 XROM "VA"
20 TONE 0
21 GTO 13
22♦LBL "ENTER"
23 RTN
24 "LBL: NAME?"
25♦LBL "?"
26 0
27 GTO 11
28♦LBL "BARRY"
29♦LBL "SHARYL"
30 .9536669
31♦LBL 11
32 CF 25
33 "  DIALING"
34 XROM "VA"
35 ENTER↑
36♦LBL 12
37 X>Y?
38 X<>Y
39 10
40 *
41 ENTER↑
42 FRC
43 GTO IND Y
44♦LBL 00
45 XROM "PO"
46 TONE 9
47♦LBL 09
48 XROM "PO"
49 TONE 9
50♦LBL 08
51 XROM "PO"
52 TONE 9
53♦LBL 07
54 XROM "PO"
55 TONE 9
56♦LBL 06
57 XROM "PO"
58 TONE 9
59♦LBL 05
60 XROM "PO"
61 TONE 9
62♦LBL 04
63 XROM "PO"
64 TONE 9
65♦LBL 03
66 XROM "PO"
67 TONE 9
68♦LBL 02
69 XROM "PO"
70 TONE 9
71♦LBL 01
72 XROM "PO"
73 TONE 9
74 ISG 00
75 GTO 12
76 GTO 13
77 RTN
```

Routine Listing For: **TN**

```
118♦LBL "TN"
119 ""
120 XROM "DC"
121 "⊢"
122 ASTO T
123 SF 25
124 "Γ♦"
125 XROM "XE"
126 CF 25
127 RTN
```

LINE BY LINE ANALYSIS OF TN

The ten lines of TN provide an excellent demonstration and training on the memory organization of the HP-41. The power of synthetic programming is evident by these lines to execute a single instruction, TONE NNN.

Line 119 prints as a double quote. This usually means that the text line contains characters that are from the second half of the HP-41 HEX table. Another indication of this is the indentation of the text line in a NORM mode printed program. All ROM routines are printed in this mode for this reason. If you SST line 119, ASTO X, and execute NH, ALPHA will show: 10 00 00 00 00 00 9F. This shows that line 119 is a single byte, 159 decimal, which is the TONE instruction. A number, 0 to 127, is assumed to be in X, and the DC routine at line 120 converts the decimal number into the corresponding alpha character. This character is appended to the TONE instruction placed there by line 119. Lines 119 and 120 synthesize a tone instruction in alpha as alpha characters. Let's assume that the long, low tone 26 is in X. Line 121 is just like line 119 and if we SST, ASTO X, and XEQ NH we find 10 00 00 00 00 00 85. Hex 85 is the RTN instruction. If 26 is in X, and lines 119 thru 121 are executed and converted to HEX using NH, the alpha register would contain: 10 00 00 00 9F 1A 85. This two instruction program is stored in the T register at line 122. The system error flag is set at line 123 "just in case" to avoid any "problems". Line 124 contains RAM address of the last three bytes of the T register. This is required for the XE routine to use to place the program pointer into the T register to execute the synthetic tone. XE was originally written for executing external ROM's, however, if the first nybble of the two byte address is 0 (the T register) then the entry is into user RAM with an address format different from that of the program pointer. See RT. The byte to be addressed is byte 3 of register 0, which makes the address 0600. Line 126 clears flag 25 and the routine is completed with the RTN at line 127. This routine is a classic example of how PPC members have mastered the HP-41. Few users outside of PPC would believe that a program could "write" a program in the stack and then exeucte it. The T register was chosen by Roger Hill (4940), so that the next input to the stack would "bump" it off the stack avoiding the potential problems of the "garbage" three bytes.

REFERENCES FOR TN

"Original" Tone article. See PPC CJ, V7n1P17c and V7N2P46b (part II). Switched QUAD information may be found in PPC J, V6N7P24b and PPC CJ, V8N1P25c.

CONTRIBUTORS HISTORY FOR TN

Roger Hill (4940) wrote TN for the PPC ROM using the concept of executing a TONE "mini-program" conceived by David Keith (5825). When the final selection for ROM routines was being made a strong consideration was made to "avoid wasting bytes on dumb tones". When PS was being planned, it was felt that as many controls for switching QUAD's as possible should be included. TN was the compromise that gave PS full TONE control and provide a convenient means to demonstrate the synthetic TONES. Multi TONE programs use two bytes for each TONE and many bytes were saved by omitting the proposed BEEP alternative programs. Only TN and T1 survived. T1 is an example of what can be done with the synthetic tones.

FINAL REMARKS FOR TN

Four routines were planned for the ROM that were best implemented in Assembly Language, or microcode. They were NS - Non-normalized Store, NR - Non-normalized Recall, VP - Variable Pause, and TP - TONE, Programmable. The SDS I System available to us did not allow these routines to be merged in. Today we can easily implement them in EPROM. The effort spent by the user community should demonstrate the high interest in greater sound output capability.

Hewlett-Packard intentionally did not provide musical TONES for the HP-41. Synthetic TONES do not come much closer to having a musical scale, but the use of "audio" has been proven to be of great practical value. The synthetic TONE capability of the PPC ROM will give this capability to all users. This should be adequate to demonstrate to Hewlett-Packard that TONES should be included in an expanded form in future personal computers.

FURTHER ASSISTANCE ON TN

Call Richard J. Nelson (1) at (714) 754-6226 P.M.
Call Roger Hill (4940) at (618) 656-8825.

TECHNICAL DETAILS

XROM: 10,32	TN	SIZE: 000

Stack Usage:	**Flag Usage:** NONE **
0 T: used (mini-program)*	04:
1 Z: saved in Y (-0)	05:
2 Y: saved in X	06:
3 X: input tone number	07:
4 L: Int(x) mod 256+256	08:
Alpha Register Usage:	09:
5 M: USED	10:
6 N: USED	14: cleared
7 O: USED	25: cleared
8 P: USED	26: set to hear tone
Other Status Registers:	**Display Mode:** N/A
9 Q:	
10 ⊦:	
11 a: NONE USED BY	**Angular Mode:** N/A
12 b: THIS ROUTINE**	
13 c:	
14 d:	**Unused Subroutine Levels:**
15 e:	5
ΣREG: NOT USED	**Global Labels Called:** Direct: DC, XE; Secondary: NONE
Data Registers: NONE USED BY ROUTINE	**Local Labels In This Routine:** NONE
Execution Time: 1.9 to 6.6 seconds.***	
Peripherals Required: NONE	
Interruptible? YES	**Other Comments:**
Execute Anytime? NO	* TONE executed in the T register.
Program File: VM	** Used by DC & XE. See these routines.
Bytes In RAM: 27	*** TONE 32 to TONE 26. Followed by CX --do not accidentally push R/S after executing TN from the keyboard.
Registers To Copy: 60	

HP-41C/CV SYNTHETIC TONE TABLE SHOWING FREQUENCY AND DURATION OF TONES 1 THRU 127

BINARY	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
HEXADEC.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
FREQ.	0	1	2	3	4	5	6	7	8	9	-6	-5	-4	-3	-2	-1
TONE NO. DISP.(P)[1] XROM DURATION 41C INST.	0 TONE 0 60,00 .28 NULL	1 TONE 1 60,01 .28 LBL 00	2 TONE 2 60,02 .28 LBL 01	3 TONE 3 60,03 .28 LBL 02	4 TONE 4 60,04 .28 LBL 03	5 TONE 5 60,05 .28 LBL 04	6 TONE 6 60,06 .28 LBL 05	7 TONE 7 60,07 .28 LBL 06	8 TONE 8 60,08 .28 LBL 07	9 TONE 9 60,09 .28 LBL 08	10 TONE 0 60,10 2.20 LBL 09	11 TONE 1 60,11 2.70 LBL 10	12 TONE 2 60,12 3.50 LBL 11	13 TONE 3 60,13 .80 LBL 12	14 TONE 4 60,14 2.3 LBL 13	15 TONE 5 60,15 .35 LBL 14
0001 1	16 TONE 6 60,16 2.00 0	17 TONE 7 60,17 .34 1	18 TONE 8 60,18 1.50 2	19 TONE 9 60,19 .33 3	20 TONE 0 60,20 .50 4	21 TONE 1 60,21 1.00 5	22 TONE 2 60,22 .45 6	23 TONE 3 60,23 .84 7	24 TONE 4 60,24 .30 8	25 TONE 5 60,25 .55 9	26 TONE 6 60,26 5.00 •	27 TONE 7 60,27 3.50 EEX	28 TONE 8 60,28 2.00 CHS	29 TONE 9 60,29 4.10 GTO α	30 TONE 0 60,30 .30 XEQ α	31 TONE 1 60,31 2.40 SPARE
0010 2	32 TONE 2 60,32 .025 RCL 00	33 TONE 3 60,33 1.13 RCL 01	34 TONE 4 60,34 2.35 RCL 02	35 TONE 5 60,35 2.00 RCO 03	36 TONE 6 60,36 1.25 RCL 04	37 TONE 7 60,37 .023 RCL 05	38 TONE 8 60,38 .023 RCL 06	39 TONE 9 60,39 .35 RCO 07	40 TONE 0 60,40 .70 RCL 08	41 TONE 1 60,41 .52 RCL 09	42 TONE 2 60,42 .85 RCL 10	43 TONE 3 60,43 .45 RCL 11	44 TONE 4 60,44 3.20 RCL 12	45 TONE 5 60,45 1.8 RCL 13	46 TONE 6 60,46 1.36 RCL 14	47 TONE 7 60,47 .13 RCL 15
0011 3	48 TONE 8 60,48 .54 STO 00	49 TONE 9 60,49 .27 STO 01	50 TONE 0 60,50 2.10 STO 02	51 TONE 1 60,51 1.95 STO 03	52 TONE 2 60,52 .28 STO 04	53 TONE 3 60,53 .15 STO 05	54 TONE 4 60,54 .80 STO 06	55 TONE 5 60,55 .77 STO 07	56 TONE 6 60,56 .65 STO 08	57 TONE 7 60,57 .058 STO 09	58 TONE 8 60,58 .4.2 STO 10	59 TONE 9 60,59 .41 STO 11	60 TONE 0 60,60 3.30 STO 12	61 TONE 1 60,61 .39 STO 13	62 TONE 2 60,62 .97 STO 14	63 TONE 3 60,63 .30 STO 15
0100 4	64 TONE 4 61,00 1.88 +	65 TONE 5 61,01 2.35 -	66 TONE 6 61,02 .40 X	67 TONE 7 61,03 .24 ÷	68 TONE 8 61,04 1.05 X<Y?	69 TONE 9 61,05 .29 X>Y?	70 TONE 0 61,06 .032 X<=Y?	71 TONE 1 61,07 .24 Σ+	72 TONE 2 61,08 .14 Σ-	73 TONE 3 61,09 .15 HMS+	74 TONE 4 61,10 3.70 HMS-	75 TONE 5 61,11 .30 MOD	76 TONE 6 61,12 3.76 %	77 TONE 7 61,13 3.40 %CH	78 TONE 8 61,14 .89 P-R	79 TONE 9 61,15 .90 R-P
0101 5	80 TONE 0 61,16 .085 LN	81 TONE 1 61,17 .22 X^2	82 TONE 2 61,18 1.75 $\sqrt{x}$	83 TONE 3 61,19 .74 y^x	84 TONE 4 61,20 .28 CHS	85 TONE 5 61,21 1.25 e^x	86 TONE 6 61,22 .50 LOG	87 TONE 7 61,23 .14 10^x	88 TONE 8 61,24 .58 e↑x-1	89 TONE 9 61,25 .050 SIN	90 TONE 0 61,26 2.70 COS	91 TONE 1 61,27 .42 TAN	92 TONE 2 61,28 3.21 ASIN	93 TONE 3 61,29 3.95 ACOS	94 TONE 4 61,30 .30 ATAN	95 TONE 5 61,31 2.40 DEC
0110 6	96 TONE 6 61,32 .65 1/x	97 TONE 7 61,33 2.32 ABS	98 TONE 8 61,34 .43 FACT	99 TONE 9 61,35 1.25 X≠0?	100 TONE 0 61,36 .12 X>0?	101 TONE 1 61,37 1. LN1+X	102 TONE A 61,38 .99 X<0?	103 TONE B 61,39 .84 X=0?	104 TONE C 61,40 .70 INT	105 TONE D 61.41 .52 FRAC	106 TONE E 61,42 .23 D-R	107 TONE F 61,43 .45 R-D	108 TONE G 61,44 3.62 HMS	109 TONE H 61,45 .33 HR	110 TONE I 61,46 2.10 RND	111 TONE J 61,47 .35 OCT
0111 7	112 TONE T 61,48 1.70 CLΣ	113 TONE Z 61,49 .65 X<>Y	114 TONE Y 61,50 1.45 PI	115 TONE X 61,51 .52 CLST	116 TONE L 61,52 1.25 R↑	117 TONEM([) 61,53 1.30 RDN	118 TONEN(\) 61,54 .24 LAST X	119 TONEO(]) 61,55 .84 CLX	120 TONEP(↑) 61,56 .14 X=Y?	121 TONEQ(_) 61,57 .33 X≠Y?	122 TONE⊢(ᵀ) 61,58 .25 SIGN	123 TONE a 61,59 4.6 X<=0?	124 TONE b 61,60 .76 MEAN	125 TONE c 61,61 4.00 SDEV	126 TONE d 61,62 3.50 AVIEW	127 TONE e 61,63 2.90 CLD

UD - UNCOVER DATA REGISTERS

This routine uses information stored in R00 by **HD** to return the curtain to the position it had prior to the last call on **HD**.

Example 1: XEQ **UD** lowers the curtain by 6 registers, if the last call on **HD** was 6, XEQ **HD**.

BACKGROUND FOR UD

See Appendix M on Curtain Moving.

COMPLETE INSTRUCTIONS FOR UD

See **HD**. **UD** preserves all of the stack, but the alpha register is used.

WARNING: **HD** and **UD** are meant to be used together in running programs. If you edit or pack program memory after using **HD** and before using **UD**, don't use **UD**. That could cause loss of catalog 1. Lower curtain with **CU** instead. **UD** assumes **HD** was executed.

LINE BY LINE ANALYSIS OF UD

Regard the initial contents of status register c as the following 14 hex digits:

$$s_1s_2\ s_30\ 01\ 69\ z_1z_2\ z_3e_1\ e_2e_3$$

where: $s_1s_2s_3$ = the abs. 3 hex-digit address of first Σ-register
$z_1z_2z_3$ = the abs. 3 hex-digit address of R00
$e_1e_2e_3$ = the abs. 3 hex-digit address of reg. containing .END.

Lines 72-75 result in moving the contents of R00

$$10\ 10\ 01\ 69\ z_1z_2\ z_3e_1\ e_2e_3$$

to status register c. The last 5½ bytes of c at the time of entry to the last execution of **HD** are therby restored. Line 76 ensures that the statistical register block lies above the established curtain: the contents of status register c upon exit is

$$z_1'z_2'\ z_3'0\ 01\ 69\ z_1z_2\ z_3e_1\ e_2e_3\quad (\overline{z_1'z_2'z_3'} = \overline{z_1z_2z_3} + 1)$$

Note that the use of **HD** / **UD** probably changes the location of the statistical register block for the calling program.

REFERENCES FOR UD

See **HD**.

CONTRIBUTORS HISTORY FOR UD

See **HD**.

FURTHER ASSISTANCE ON UD

Call Harry Bertuccelli (3994) at (213) 846-6390.
Call Keith Jarett (4360) at (213) 374-2583.

TECHNICAL DETAILS

XROM: 10,08 — **UD** — SIZE: UNCHANGED SINCE EXECUTION OF **HD**

Stack Usage:
0 T: ALL UNCHANGED
1 Z:
2 Y:
3 X:
4 L:

Alpha Register Usage:
5 M: ALPHA R_{00}
6 N: CLEARED
7 O: CLEARED
8 P: CLEARED

Flag Usage: NONE USED
04:
05:
06:
07:
08:
09:
10:
25:

Other Status Registers:
9 Q: NOT USED
10 ⊢: NOT USED
11 a: NOT USE
12 b: NOT USED
13 c: CHANGED
14 d: NOT USED
15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

ΣREG: SET TO 01

Data Registers:
R00: NO REGISTERS ALTERED FORMER R00 MUST BE INITIALIZED.
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: .4 seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO!
Program File: **LF**
Bytes In RAM: 14
Registers To Copy: 59

Other Comments: **HD** MUST be used before **UD**. Otherwise you'll get MEMORY LOST.

Routine Listing For: **UD**

```
72♦LBL "UD"      75 ASTO c
73 CLA           76 ΣREG 01
74 ARCL 00       77 RTN
```

APPENDIX L CONTINUED FROM PAGE 429.

In addition, certain of the data barcodes are 7-byte text lines. These load into ALPHA in such a way that a RCL M instruction is required to bring it into X for correct accumulation into the print buffer. The other, shorter text lines may be placed into X by ASTO X, since the lines do not contain information in the first byte, which includes the nybble which is the sign of the mantissa. In the barcodes to follow, those marked 'M' require RCL M and those unmarked require ASTO X before ACSPEC.

LINE BY LINE ANALYSIS OF SC

Lines 01 through 06 determine which text line is to be placed in the X register, by a computed branch to a numeric label.

Lines 07 through 14 determine whether the text string is to be brought into the X register via RCL M (if the text line is 7 characters long) or by ASTO X (if the string is shorter than 7 characters). In addition, for the characters which may be superscripts or subscripts, 2 null bytes are appended if flag 10 is set, converting the superscript string to a subscript.

Lines 15 through 197 consist of the 57 individual subroutines which place the text lines into ALPHA.

BAR CODE ON PAGE 482

Routine Listing For: **SC**

```
01♦LBL "SC"
02 15
03 XROM "QR"
04 20
05 ST+ Z
06 XEQ IND Z
07 FS?C 10
08 "├♦♦"
09 RCL [
10 SF 25
11 CHS
12 FS?C 25
13 ASTO X
14 RTN
15♦LBL 20
16 XEQ IND Y
17 RTN
18♦LBL 00
19 "⁎H"
20 RTN
21♦LBL 01
22 "α0"
23 RTN
24♦LBL 02
25 "↓J"
26 RTN
27♦LBL 03
28 "βJ"
29 RTN
30♦LBL 04
31 "⁎Σ"
32 RTN
33♦LBL 05
34 "β"
35 RTN
36♦LBL 06
37 "↓"
38 RTN
39♦LBL 07
40 "@"
41 RTN
42♦LBL 08
43 "↓"
44 RTN
45♦LBL 09
46 "⁎"
47 RTN
48♦LBL 10
49 "Γ£"
50 RTN
51♦LBL 11
52 "←"
53 RTN
54♦LBL 12
55 "ΓJ"
56 RTN
57♦LBL 13
58 "⁎↓α"
59 RTN
60♦LBL 14
61 "⁎x̄α"
62 RTN
63♦LBL 21
64 XEQ IND Y
65 RTN
66♦LBL 00
67 "x̄x̄x̄"
68 RTN
69♦LBL 01
70 "←"
71 RTN
72♦LBL 02
73 "Δ"
74 RTN
75♦LBL 03
76 "p ♦"
77 RTN
78♦LBL 04
79 "@Σ
@"
80 RTN
81♦LBL 05
82 "x̄α↓@"
83 RTN
84♦LBL 06
85 "ΩGδōH"
86 RTN
87♦LBL 07
88 "ΩGā0ō"
89 RTN
90♦LBL 08
91 "ΩGx̄>H"
92 RTN
93♦LBL 09
94 "⁎2♦"
95 RTN
96♦LBL 10
97 "x̄$K"♦"
98 RTN
99♦LBL 11
100 " x̄Δ"
101 RTN
102♦LBL 12
103 "  ÄαΔΔ"
104 RTN
105♦LBL 13
106 "δ$ȧ$"
107 RTN
108♦LBL 14
109 "ȧQ"""
110 RTN
111♦LBL 22
112 XEQ IND Y
113 RTN
114♦LBL 00
115 "⁎R      J♦"
116 RTN
117♦LBL 01
118 "x̄
119 RTN
120♦LBL 02
121 "x̄*(♦"
122 RTN
123♦LBL 03
124 "◂ΩÄβΩe"
125 RTN
126♦LBL 04
127 "μkō,F"
128 RTN
129♦LBL 05
130 "x̄Σ?
"
131 RTN
132♦LBL 06
133 "⁎J♦"
134 RTN
135♦LBL 07
136 "@G≠⁎"
137 RTN
138♦LBL 08
139 "'αΓ♦"
140 RTN
141♦LBL 09
142 "μaΣΓ←"
143 RTN
144♦LBL 10
145 "$↓0"8"
146 RTN
147♦LBL 11
148 "◂Ä
Qe"
149 RTN
150♦LBL 12
151 "α├μαx"
152 RTN
153♦LBL 13
154 "Ωμ00ᵀ"
155 RTN
156♦LBL 14
157 "ΩμX4ᵀ"
158 RTN
159♦LBL 23
160 XEQ IND Y
161 RTN
162♦LBL 00
163 "ΩμY4ᵀ"
164 RTN
165♦LBL 01
166 "Ω∡X5ᵀ"
167 RTN
168♦LBL 02
169 "Ω∡Y5ᵀ"
170 RTN
171♦LBL 03
172 "Ω∡7ᵀ"
173 RTN
174♦LBL 04
175 " Δ"
176 RTN
177♦LBL 05
178 "8"
179 RTN
180♦LBL 06
181 "8"
182 RTN
183♦LBL 07
184 "pe"
185 RTN
186♦LBL 08
187 "⁎ "
188 RTN
189♦LBL 09
190 "Ω♦♦"
191 RTN
192♦LBL 10
193 "⁎H"
194 RTN
195♦LBL 11
196 "ΩΓ♦$G♦♦"
197 END
```

REFERENCES FOR SC

See PPC Calculator Journal, V7N10P11b.

APPENDIX L CONTINUED ON PAGE 443.

UR - UNPACK REGISTER

This routine is called unpack register and can be used to recall packed data from a data register. The packing scheme is simply encoded data assuming a base b representation that is usually other than base 10. This routine first appeared on the HP-67/97 in the booklet BETTER PROGRAMMING ON THE HP-67/97. See also the routine PR. Using base b data packing techniques it is possible to store several numbers in one register. The UR routine is used to recall the packed numbers from a data register.

Example 1: If the base b=52 and the register R15 contains the number 217,499,926 then R15 may be assumed to hold 5 packed numbers all in the range 0-51. Use UR to recall those numbers from R15.

As does PR, the UR routine assumes that the base b is stored in R10 and that R11 contains the number of the data register that will be unpacked. For this example store the following data.

R10: 52 = base b
R11: 15 = pointer to register R15
R15: 217,499,926 = the packed data

To understand the packing routine consider how the number 217,499,926 appears when written in polynomial form where the base b=52.

$217{,}499{,}926 = 29*52^4+38*52^3+44*52^2+18*52^1+46*52^0$

The five packed numbers are the coefficients of the powers of 52. These five coefficients are assumed to be numbered from 1-5 starting with the zero power of 52. The powers of 52 range from 0-4 but the UR (and PR) routine assumes the corresponding range to be 1-5. In this example we can see the correspondence between the position numbers and the stored data.

The number 29 corresponds to position 5.
The number 38 corresponds to position 4.
The number 44 corresponds to position 3.
The number 18 corresponds to position 2.
The number 46 corresponds to position 1.

Provided that R10, R11, and R15 have been initialized with the above data it is a simple matter to use UR to recall the packed data.

Key in any position number from 1-5 and XEQ "UR". For example, key in 3 and XEQ "UR" and see 44 returned in X.

Example 2: Change the base in Example 1 to b=2. Then recall positions 15-20.

Key 2 STO 10. This example assumes the number 15 is still in R11 and that the number 217,499,926 is still in R15. When the base is 2 the position numbers range between 1 and 30 and in this case each register may be assumed to hold 30 binary flags.

Key in 15 and XEQ "UR". See 1. Flag 15 is set.
Key in 16 and XEQ "UR". See 1. Flag 16 is set.
Key in 17 and XEQ "UR". See 0. Flag 17 is clear.
Key in 18 and XEQ "UR". See 1. Flag 18 is set.
Key in 19 and XEQ "UR". See 1. Flag 19 is set.
Key in 20 and XEQ "UR". See 0. Flag 20 is clear.

COMPLETE INSTRUCTIONS FOR UR

1) UR assumes that R10 holds the base b and that R11 is pointing to the register that contains a number to be decoded.

R10: base b
R11: register pointer

2) To recall the number stored in position k in the register pointed to by R11, key in k and XEQ "UR". UR will return in the X-register a number in the range 0-(b-1). The following is the stack input/output for UR.

Input:		Output:	
	T: T		T: Y
	Z: Z		Z: Y
	Y: Y		Y: b^{k-1}
	X: position k		X: data number
	L: L		L: b=base in R06

The following table indicates the range of possible bases and position numbers.

Data Range	Base b	Position Numbers
0-1	2	1-30
0-2	3	1-19
0-3	4	1-15
0-4	5	1-13
0-6	7	1-11
0-9	10	1-10
0-13	14	1-8
0-20	21	1-7
0-36	37	1-6
0-99	100	1-5
0-214	215	1-4
0-1413	1414	1-3
0-99999	100000	1-2

The most efficient use may be made of data registers by storing the largest data values in the lowest numbered positions and storing the smallest data values in the highest numbered positions. If your priority is the range of data, start with the column on the left. If your priority is the number of artificial memories available, start with the column on the right. In many cases it will be possible to extend the values in this table.

MORE EXAMPLES OF UR

Example 3: See Example 2 of the PR routine where the base b=21, the register holding the packed data is R12 and the actual data is 1,447,473,103 which in base 21 form is:

$16*21^6 + 18*21^5 + 8*21^4 + 15*21^3 + 14*21^2 + 19*21 + 13$

Store the following data:

R10: 21 = base
R11: 12 = pointer to register 12
R12: 1,447,473,103 = packed form of the data

Use UR to recall the numbers in positions 1-7.

Key 1 XEQ "UR" and see 13 returned
Key 2 XEQ "UR" and see 19 returned
Key 3 XEQ "UR" and see 14 returned
Key 4 XEQ "UR" and see 15 returned
Key 5 XEQ "UR" and see 8 returned
Key 6 XEQ "UR" and see 18 returned
Key 7 XEQ "UR" and see 16 returned

Routine Listing For: **UR**

```
216♦LBL "UR"
217 1
218 -
219 RCL 10
220 X<>Y
221 Y↑X
222 RCL IND 11
223 X<>Y
224 ST/ Y
225 X<>Y
226 INT
227 RCL 10
228 MOD
229 RTN
```

LINE BY LINE ANALYSIS OF UR

Lines 217 & 218 subtract 1 from the position number which result in a number corresponding exactly to the powers of the base.

The digit corresponding to the position number is extracted by first throwing away all the higher powers of b at line 226. The desired number is then calculated at line 228.

REFERENCES FOR UR

John Kennedy, "Data Packing," BETTER PROGRAMMING ON THE HP-67/97," by Bill Kolb (265), Richard Nelson (1), and John Kennedy (918).

CONTRIBUTORS HISTORY FOR UR

UR and its corresponding documentation were written by John Kennedy (918).

FURTHER ASSISTANCE ON UR

John Kennedy (918) phone: (213) 472-3110 evenings
Richard Schwartz (2289) phone: (213) 447-6574 eve.

TECHNICAL DETAILS

XROM: 20, 44 — **UR** — SIZE: depends on data used

Stack Usage:
- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:
- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:
- 9 Q: not used
- 10 ⊦: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:
- R00: not used
- For data storage at least
- R06: One additional data register is used
- R07: by UR
- R08:
- R09:
- R10: base b
- R11: pointer to data reg.
- R12:

Flag Usage:
- 04: not used
- 05: not used
- 06: not used
- 07: not used
- 08: not used
- 09: not used
- 10: not used
- 25: not used

Display Mode: not used

Angular Mode: not used

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
none	none

Local Labels In This Routine: none

Execution Time: 1.0 seconds

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: M2

Bytes In RAM: 23

Registers To Copy: 61

Other Comments:

VA - VIEW ALPHA

The standard HP-41C function AVIEW will display the contents of the alpha register and print it if the printer is present and turned on and flag 21 is set. However, if flag 21 is set and the printer is not present execution comes to a grinding halt with alpha in the display and no TONE or error message of any kind. (If flag 21 is set and the printer is turned off, execution halts with the PRINTER OFF message.) This behavior can be useful, but it's often a nuisance, especially when you're not expecting it. The PPC ROM routine VA provides an alternative. VA displays the contents of the alpha register and prints it if the printer is present, turned on, and enabled (flag 21 set). The status of flag 21 is preserved and in no case is execution halted.

Example 1: GTO.. and key in the following lines.

```
01  "TEST"
02  XROM VA
03  BEEP
```

In RUN mode set flag 26, turn off the printer (if it is present), SF 21, RTN, and R/S. You should see "TEST" displayed and hear a BEEP. Now turn on the printer and R/S. You should see "TEST" displayed and printed and hear the BEEP. Now clear flag 21 and R/S again. You should see "TEST" displayed, but not printed, and hear a BEEP.

COMPLETE INSTRUCTIONS FOR VA

Use XROM VA everywhere you would use AVIEW, unless you want execution to halt when the printer is not present and turned on. VA does not disturb the stack or alpha register; the only status change is that flag 25 is cleared.

Do not press R/S after executing VA from the keyboard. VA is followed in the ROM by UD, certain to cause MEMORY LOST if it is run without preparation; better yet--don't execute VA manually.

MORE EXAMPLES OF VA

Example 2: The following program segment will display the contents of the alpha register and print alpha only if flag 21 is set. It behaves like VA except it does not view alpha if flag 21 is clear.

```
FS? 21
XROM VA
```

LINE BY LINE ANALYSIS OF VA

Lines 62-64 print the alpha register if the printer is turned on and enabled (flag 21 set). Line 65-67 transfer the status of flag 21 to flag 25 and clear flag 21, preventing the AVIEW on line 68 from halting execution. Lines 69-71 transfer the status of flag 25 back to flag 21, clear flag 25, and return to the calling program.

CONTRIBUTORS HISTORY FOR VA

VA was written by Roger Hill (4940) for the PPC ROM to provide a "print if possible, but never stop" capability for other ROM routines. Richard Nelson (1) was a strong proponent of this concept.

TECHNICAL DETAILS

XROM: 10,07 — **VA** — SIZE: 000

Stack Usage:		Flag Usage: ONLY FLAGS 21 and 25 USED	
0 T:	ALL UNCHANGED	04:	
1 Z:		05:	
2 Y:		06:	
3 X:		07:	
4 L:		08:	
Alpha Register Usage:		09:	
5 M:	ALL UNCHANGED	10:	
6 N:		21:	USED BUT RESTORED
7 O:		25:	CLEARED
8 P:			
Other Status Registers:	NONE USED	**Display Mode:**	UNCHANGED
9 Q:			
10 ⊦:		**Angular Mode:**	UNCHANGED
11 a:			
12 b:			
13 c:		**Unused Subroutine Levels:**	6
14 d:			
15 e:			
ΣREG:	UNCHANGED	**Global Labels Called:** Direct	Secondary
Data Registers:	NONE USED	NONE	NONE
R00:		**Local Labels In This Routine:**	NONE
R11:			
R12:			

Execution Time: .5 seconds.

Peripherals Required: NONE

		Other Comments:
Interruptible?	YES	
Execute Anytime?	YES	
Program File:	LF	
Bytes In RAM:	22	
Registers To Copy:	59	

Routine Listing For: VA

```
62♦LBL "VA"      67 CF 25
63 SF 25         68 AVIEW
64 PRA           69 FC?C 25
65 SF 25         70 SF 21
66 FS?C 21       71 RTN
```

FURTHER ASSISTANCE ON VA

Call Roger Hill (4940) at (618) 656-8825.
Call Richard Nelson (1) at (714) 754-6226.

APPENDIX L CONTINUED FROM PAGE 439.

SUPERSCRIPTS	Code	Character	SUBSCRIPTS
	0	o	
	1	1	
	2	2	
	3	3	
	4	4	
	5	5	
	6	6	
	7	7	
	8	8	
	9	9	
	10	X	
	11	Y	
	12	Z	
	13	+	
*	14	-	*
*	15	⁄	*
	16	(	
	17	)	

Code	Character	Note
18	▫	
19	√	
20	∫	
21	dx	M*
22	dy	M*
23	dt	M*
24	∂	
25	±	
26	↚	
27	~	
28	≃	
29	≈	
30	≡	
31	≤	
32	≥	
33	∞	
34	∇	
35	Π	*
36	∈	
37	σ	
38	ν	
39	Ψ	
40	ω	
41	®	
42	™	
43	⚀	M*
44	⚁	M*
45	⚂	M*
46	⚃	M*
47	⚄	M*
48	⚅	M*
49	♦	
50	♥	
51	♠	
52	♣	
53	◀	
54	▶	M*
55	⊂	
56	⊃	M*

APPENDIX L CONTINUED ON PAGE 449.

VF - VIEW FLAGS

VF provides a capability somewhat similar to the printer function PRFLAGS. It tells the user the status of all 56 flags in a compact format.

Example 1: Display the status of the flags after RF. XEQ RF then XEQ VF. See "FLAGS SET:", then "26 28 29 38" and "40 50". A BEEP signals completion. These are the RF flag settings except for flag 50, the message flag. VF always shows flag 50 set since there is always a message in the display by the time flag 50 is checked.

COMPLETE INSTRUCTIONS FOR VF

Just XEQ VF to display the flag status. The status will also be printed out if the printer is turned on and enabled. Flag 50 will always show as set.

The contents of X and Y are saved, a temporary flag register (from line 50) is placed in T, a number from 1 to 4 is placed in Z, and 56.055 is placed in L. Alpha contains the last part of the flag status message.

MORE EXAMPLES OF VF

Example 2: Determine the flag status when the annunciator test for DT is executed. XEQ DT and switch out of PRGM mode when the annunciators light. Then XEQ VF. The display will show "FLAGS SET:", "0 1 2 3", "4 27 42 49", "50". This display does not include the shift flag (47), the alpha mode flag (48), and the PRGM mode flag (52), which were lost when you switched out of PRGM mode.

LINE BY LINE ANALYSIS OF VF

Basically the routine tests each flag and, if it is set, appends that flag number to the alpha register, displaying and/or printing the flag numbers whenever four of them have accumulated in alpha. There are a few problems however, that have to be dealt with in implementing this procedure. Displaying and printing the flag numbers, as well as sounding a tone, require certain flag settings, and to avoid disturbing the original flags we temporarily exchange the original flag register with a new set of flags whenever alpha-recalling or viewing is done. An exception to the non-disturbance of the original flags is flag 50, which we set at the beginning (lines 44 through 46) to keep the display from scrolling during the flag search loop. Flag 50 will therefore, always show up in the list of flags set, but this is not likely to be a hardship in practice.

The routine begins by setting flag 50 as just described by synthetic means, and then creating the number 00 05 50 00 00 00 00, in putting it in register L (lines 47 through 49). This number is simply the flag index 0.055 to be used with ISG, but in the non-normalized form 0.055 x 10^0 rather than the usual form 5.5 x 10^{-2} in which the number would be stored if created in X. When ARCL'd in FIX 0, CF 29 format, the former shows up as 0 (as desired), whereas the latter would show up as 6.E-2 instead. In lines 50 and 51 we create and put in X a set of temporary flags, in which flags 21, 24, 26, 28, 40, and 55 are set (flag 50 will also get set in line 56). These set the appropriate display format, enabling the printer (in case if it is used) and the tones, and also set flag 24, the clearing of which will mean that there is a final row of 1 to 4 flag

TECHNICAL DETAILS

XROM: 20,58 — **VF** — SIZE: 000

Stack Usage:
- 0 T: temporary d
- 1 Z: 1,2,3, or 4
- 2 Y: UNCHANGED
- 3 X: UNCHANGED
- 4 L: 56.055

Alpha Register Usage:
- 5 M:
- 6 N: FLAG NUMBER
- 7 O: DISPLAY
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Flag Usage: SEVERAL USED BUT ALL RESTORED
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
IF	NONE
VA	

Local Labels In This Routine:
01
02
03

Execution Time: 11.8 - 37.0 seconds.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? YES
Program File: SM
Bytes In RAM: 101 WITH END
Registers To Copy: 26

Other Comments:

VF

numbers in alpha waiting to be viewed. The number 4 is placed in T (lines 52 through 53) to be used as a counter to determine when four flag numbers have accumulated in alpha. We then view the "FLAGS SET:" heading in lines 55 through 58 (using the new flags to make sure flag 21 is set) and are ready for the main flag search loop, lines 59 through 63.

During the flag search loop the stack is as follows: T = 4-(number of flag numbers in alpha), Z = original Y, Y = original X, X = temporary flags, L = flag index. When a set flag is found we go to lines 72 through 87 to put the flag number in alpha (lines 73 through 75, clear flag 24 (line 76), and check to see if four flag numbers have accumulated. If not, (i.e., T has not reached zero) we return to the flag search loop via lines 85 through 87. If so, we view and/or print them, sound a Tone 6, clear alpha, and reset the counter in T and flag 24 (lines 79 through 84).

When the flags have all been tested, we use flag 24 to decide whether there are some flag numbers in alpha that have not yet been viewed, and view them if there are (lines 64 through 66). We then finish up by advancing a line, beeping to indicate the end, restoring the flags, and restoring the X and Y of the stack before returning (lines 67 through 71).

CONTRIBUTORS HISTORY FOR VF

A crude version of VF (see *PPC CALCULATOR JOURNAL*, V7N7P17) was written by Keith Jarett (4360) for the first compilation of ROM routines. Roger Hill (4940) managed to come through with a usable version in time for the ROM loading.

FURTHER ASSISTANCE ON VF

Call Roger Hill (4940) at (618) 656-8825.
Call Richard Nelson (1) at (714) 754-6226.

Routine Listing For: **VF**

```
43♦LBL "VF"
44 50
45 FC? 50
46 XROM "IF"
47 "ΒP♦♦♦♦"
48 RCL [
49 SIGN
50 "α        ♦x"
51 X<> [
52 4
53 RDN
54 "FLAGS SET:"
55 X<> d
56 XROM "VA"
57 X<> d
58 CLA

59♦LBL 01
60 FS? IND L
61 XEQ 02
62 ISG L
63 GTO 01
64 X<> d
65 FC?C 24
66 XROM "VA"
67 ADV
68 BEEP
69 X<> d
70 RDN
71 RTN

72♦LBL 02
73 "├ "
74 X<> d
75 ARCL L
76 CF 24
77 DSE T
78 GTO 03
79 XROM "VA"
80 TONE 6
81 CLA
82 4
83 RDN
84 SF 24

85♦LBL 03
86 X<> d
87 END
```

NOTES

VK - VIEW KEY ASSIGNMENTS

PRKEYS (Print Key Assignments) is an 82143A Printer Function which prints the keycode of the reassigned key followed by the name of the program or function assigned to that key. VK gives a somewhat similar capability to the HP-41C mainframe without the need for a printer. VK identifies those keys which have assignments by displaying their keycodes.

Example 1: Make the following key assignments

```
USER KEYS:
 11 MEAN
-11 SDEV
 12 INT
-12 FRC
 13 ABS
-13 RND
 14 MOD
-14 SIGN
 42 DSE
-44 CLST
 84 PSE
-84 ADV
```

Unplug the printer (if present), XEQ VK and see displayed

```
11  -11
12  -12
13  -13
14  -14
42
    -44
84  -84
```

Before executing VK again, do the following

1) SIZE 000.
2) Turn USER mode on.
3) XEQ "GRAD".
4) SF 00, SF 02 SF 04.
5) SCI 5.

Now XEQ VK again and see the same keycodes displayed as above. Note that VK executed correctly with SIZE 000. Additionally, note that it apparently does not change modes/flags. Register d is used by VK , but its original contents are restored before a normal program halt.

COMPLETE INSTRUCTIONS FOR VK

VK has the following characteristics:

1) It displays keycodes if printer is not present.
2) It defaults to PRKEYS if printer is present.
3) It preserves user's original mode/flag settings.
4) It can be interrupted, but another R/S is required to "clean up" after interruption.
5) It leaves the stack and alpha register clear.

MORE EXAMPLES OF VK

Example 2: XEQ VK again and press R/S before the "84 -84" appears. The ALPHA mode will probably be on, as well as other miscellaneous annunciators/flags. Press R/S again and the original modes/flags are restored.

Example 3: If you have a printer available, install it. XEQ VK and note that PRKEYS is executed.

```
USER KEYS:
 11 MEAN          14 MOD
-11 SDEV         -14 SIGN
 12 INT           42 DSE
-12 FRC          -44 CLST
 13 ABS           84 PSE
-13 RND          -84 ADV
```

Example 4: XEQ CK (PPC ROM ROUTINE) to clear the key assignments which were made above. Now XEQ VK one more time. The "flying goose" stops at the third character position and remains there approximately 15 seconds while VK determines that there are no key assignments.

Example 5: The sequence

```
55
FS? 55
XROM IF
XROM VK
```

allows VK to be run in a program regardless of whether the printer is present.

LINE BY LINE ANALYSIS OF VK

VK first tries to execute the printer function PRKEYS if flag 55 is set (normally meaning that the printer is present). If PRKEYS succeeds (printer on), the RTN at line 06 is executed.

If flag 55 is clear (normally meaning that the printer is not present), PRKEYS is bypassed and flag 21 is cleared to allow the later use of AVIEW. With flag 55 clear, status register Q can safely be used as a scratch register.

Lines 09 thru 10 use a subroutine (lines 113 thru 119) to isolate the first five bytes of the ⊦ register in M. Lines 11 thru 13 append two identifier bytes (hex 10 F0) for the unshifted flags and exchange them with the system flags. Lines 14 thru 16 isolate the first five bytes of the e register and append hex 20 F0 to identify the shifted flags. Lines 17 and 18 place 8 in M and put the shifted flags in X. The unshifted flags remain in d; the system flags are in Y.

LBL 01 (lines 19 thru 24) begin the processing of a new row, constructing the ISG flag counter to enable flags to be checked in the correct order from column 1 to column 5. For row 1 this column control number is -35.00008; for row 8 it's -28.00008. The column control number is placed in N. Lines 25-36 examine each column in turn. First the unshifted flags are tested (key 11 = flag 35, etc.), then the shifted flags swapped into d and tested. The unshifted flags are then put back in d for the next time through the column loop. Lines 37 and 38 complete the row examination (outer) loop.

If a flag is found set on line 26 or 30, the keycode display forming part of VK is entered at LBL 05. Lines 49 thru 63 convert the flag number to the keycode and store it in status register a. Lines 64 thru 69 adjust row 4 key codes, while lines 70 thru 73 place +1 or -1 in L depending on whether one or both shifted and unshifted flags are set.

BAR CODE ON PAGE 483

ALTERNATE VERSION OF: VK

```
01♦LBL "VK"
02 8
03 STO a
04 SF 25
05 PRBUF
06 FC? 25
07 CF 21
08 "KEYS USED:"
09 AVIEW
10 "8α♦♦♦♦♦♦"
11 RCL '
12 X<> [
13 RCL e
14 X<> \
15 STO ]
16 FIX 3
17 CF 29
18 ARCL Y
19 FIX 0
20 X<> ]
21 STO [
22 "├ "
23 X<> \
24 X<> d
25 RCL [
26 CLA

27♦LBL 01
28 X<> Z
29 ABS
30 FRC
31 CHS
32 LASTX
33 INT
34 +
35 39
36 -
37 X<> Z

38♦LBL 02
39 FC? IND Z
40 FC? 50
41 GTO 15
42 X<> d
43 FC? IND Z
44 FC? 50
45 GTO 15
46 X<> d

47♦LBL 03
48 ISG Z
49 GTO 02
50 DSE a
51 GTO 01
52 X<>Y
53 STO d
54 FS? 21
55 CLA
56 "├ END"
57 AVIEW
58 GTO 14

59♦LBL 15
60 FC? 50
61 GTO 15
62 X<> d
63 STO \
64 X<> Z
65 STO [
66 INT
67 123
68 +
69 8
70 /
71 INT
72 LASTX
73 FRC
74 80
75 *
76 +
77 41
78 X<Y?
79 DSE Y
80 3
81 +
82 X<Y?
83 ISG Y
84 ""
85 FS? 42
86 FC? IND [
87 CHS
88 X<>Y
89 ABS
90 X<> [
91 R↑
92 RCL \
93 " "
94 FC? 42
95 "    -"
96 X<>Y
97 X<> d
98 ARCL L
99 ISG T
100 GTO 13
101 "├ -"
102 ARCL L

103♦LBL 13
104 AVIEW
105 FC? 21
106 TONE 0
107 X<> d
108 X<>Y
109 FS? 42
110 X<> d
111 GTO 03

112♦LBL 15
113 RDN
114 STO d
115 CLD

116♦LBL 14
117 FC?C 25
118 SF 21
119 CLST
120 FIX 2
121 END
```

Routine Listing For: VK

```
01♦LBL "VK"
02 SF 21
03 FS? 55
04 PRKEYS
05 FS? 55
06 RTN
07 CF 21
08 8
09 RCL '
10 XEQ 07
11 "├8"
12 X<> [
13 X<> d
14 RCL e
15 XEQ 07
16 "├ "
17 X<> Z
18 X<> [

19♦LBL 01
20 -27.00008
21 RCL [
22 -
23 STO \
24 RDN

25♦LBL 02
26 FC? IND \
27 FC? 50
28 GTO 05
29 X<> d
30 FC? IND \
31 FC? 50
32 GTO 05
33 X<> d

34♦LBL 03
35 ISG \
36 GTO 02
37 DSE [
38 GTO 01
39 X<>Y

40♦LBL 04
41 STO d
42 CLST
43 CLA
44 PSE
45 CLD
46 RTN

47♦LBL 05
48 X<> d
49 35
50 RCL \
51 INT
52 +
53 OCT
54 1
55 ST+ Y
56 %
57 +
58 10
59 MOD
60 LASTX
61 *
62 INT
63 STO a
64 43
65 -
66 ABS
67 1
68 X<Y?
69 ST+ a
70 FS? 42
71 FC? IND \
72 CHS
73 ABS
74 X<> [
75 RDN
76 X<> \
77 RDN
78 " "
79 FC? 42
80 "    -"
81 FS? 50
82 X<> d
83 X<>Y
84 FC? 50
85 GTO 04
86 X<> d
87 X<> _
88 CLX
89 RCL d
90 FIX 0
91 CF 29
92 ARCL a
93 ISG L
94 GTO 06
95 "├ -"
96 ARCL a

97♦LBL 06
98 STO d
99 X<> _
100 AVIEW
101 TONE 0
102 R↑
103 STO \
104 R↑
105 STO [
106 RDN
107 RDN
108 X<> d
109 X<>Y
110 FS? 42
111 X<> d
112 GTO 03

113♦LBL 07
114 CLA
115 X<> [
116 "├*****"
117 X<> \
118 X<> [
119 RTN
```

TECHNICAL DETAILS

XROM: 10,36	VK	SIZE: 000

Stack Usage: 0 T: CLEARED 1 Z: CLEARED 2 Y: CLEARED 3 X: CLEARED 4 L: USED	**Flag Usage:** SEVERAL USED BUT ALL RESTORED 04: 05: 06: 07: 08:
Alpha Register Usage: 5 M: 6 N: ALL CLEARED 7 O: 8 P:	09: 10: 21: SET IF PRINTER PRESENT CLEARED OTHERWISE 25:
Other Status Registers: 9 Q: USED 10 ⊢: UNCHANGED 11 a: USED 12 b: NOT USED 13 c: NOT USED 14 d: USED BUT RESTORED 15 e: UNCHANGED	**Display Mode:** UNCHANGED **Angular Mode:** UNCHANGED **Unused Subroutine Levels:** 2
ΣREG: UNCHANGED **Data Registers:** NONE USED R00: R06: R07: R08: R09: R10: R11: R12:	**Global Labels Called:** Direct: NONE Secondary: NONE **Local Labels In This Routine:** 01 02 03 04 05 06 07
Execution Time: 15½ - 87 seconds.	
Peripherals Required: NONE	
Interruptible? YES Execute Anytime? YES Program File: VK Bytes In RAM: 222 Registers To Copy: 63	**Other Comments:** CLD before calling in a program (to clear flag 50)

Lines 74 thru 77 save the loop control numbers from M and N in the stack so that the AVIEW can be performed. Lines 78 thru 80 form the first part of the display, which carries a minus sign if the shifted flag got you to LBL 05 (meaning the unshifted flag was clear). Lines 81 thru 85 align the stack and exit (LBL 04) if the program has been interrupted (flag 50 clear). Lines 86 thru 92 save the shifted or unshifted flags in status register Q, put the system flags in register d, and attach the keycode. Unless L = -1, meaning that both shifted and unshifted flags are set, the AVIEW procedure is begun (LBL 06). If L = -1 the shifted keycode is appended before dropping into LBL 06.

LBL 06 restores the display mode (with flag 21 clear), extracts the shifted or unshifted flags from Q, performs the AVIEW, and sounds TONE 70. Lines 102 thru 107 restore the loop control numbers to M and N and realign the stack. Lines 108 thru 112 put the system flags in Y, the shifted flags in X, and the unshifted flags in d. Control returns to the bottom of the flag number loop. When all 35 flags have been checked, we fall into the LBL 04 cleanup sequence.

Program Notes:

1) Line 10 is hex F9 10 F0 04 00 00 80 00 00 00.
2) Line 22 is hex F3 7F 20 F0.
3) Lines 56, 93, 95, and 101 are not synthetic.
4) Line 106 is 9F 46 (TONE 70).

CONTRIBUTORS HISTORY FOR VK

The first version of VK (PPC CJ, V7N7P18) was written by Roger Hill (4940) and called KU (Keys Used), (see PPC CJ, V7N10P18). It required SIZE 004. Richard Collett (4523) and Tom Cadwallader (3502) both wrote SIZE 000 versions. Richard's version (listed below for reference) produces exactly the same output ("KEYS USED", "END", etc.) as Roger's original KU, whether or not a printer is present. This may be more to the liking of some users. The ROM Committee almost had to use RN to make the choice of VK.

FINAL REMARKS FOR VK

VK allows users to conveniently view all key assignments. For example, when MK produces the message "KEY TAKEN", VK can be executed without destroying MK's data base. XEQ VK, choose an unused key, and XEQ +K to continue with MK.

CK, MK, PK, and VK form a complete set of key assignment routines. They clear, make, pack, and view key assignments, respectively.

FURTHER ASSISTANCE ON VK

Call Tom Cadwallader (3502) at (406) 727-6869.
Call Roger Hill (4940) at (618) 656-8825.

APPENDIX L CONTINUED FROM PAGE 443.

CONTRIBUTORS HISTORY FOR SC

The SC program was originally writted by Jake Schwartz (1820), and was modified by Roger Hill (4940) to reduce the byte count significantly. Additional assistance for choice and design of the special printer characters was provided by John McGechie (3324), Earnest Gibbs (4610), William Wimsatt (5807) and Randall Pratt (2860).

FINAL REMARKS FOR SC

This program exemplifies the value of the wand as a device for creation of 41C synthetic program lines. In conjunction with the printer, the wand and barcode can provide new character sets for almost any special application. The characters chosen for the SC program were those which were felt to be most useful to the people who would have the PPC ROM. When the PPC Barcode Book is produced, we will be able to further exploit the advantages of scanning synthetic text lines dirtectly into HP41C program memory.

Synthetic text lines in the SC program:

```
08: Append 2 nulls       46: F3 01 C2 9F
19: F4 01 C4 48 8E       49: F3 06 C2 1B
22: F3 04 4F 90          52: F2 CE 03
25: F3 07 4A 97          55: F3 06 4A 93
28: F3 05 4A 9F          58: F3 01 07 04
34: F3 01 C2 1F          61: F3 01 02 04
34: F3 05 CA 9D          67: F3 02 02 02
37: F3 07 CA 9D          70: F3 03 88 80
40: F2 40 9F             73: F2 08 8E
43: F3 07 CA 9F          76: F4 70 A1 C0 00
79: F6 02 04 07 C0 40 80
82: F6 40 82 0F E0 40 81
85: F7 11 E2 47 F0 12 18 48
88: F7 11 E2 47 F0 16 10 18
91: F7 11 E2 47 F0 02 3E 48
94: F6 01 8C 99 32 9E 00
97: F6 02 24 4B F1 22 00
100: F6 20 C2 81 02 86 08
103: F6 20 20 41 04 08 08
106: F6 91 12 24 8A 14 24
109: F6 88 89 14 51 22 22
115: F6 01 52 A5 4A 95 00
118: F6 02 85 8A 94 A8 80
121: F6 02 8D 2A 96 28 00
124: F6 71 11 41 05 11 1C
127: F6 0C 6B 18 2C 46 83
130: F6 02 0F F8 3F E0 80
133: F5 01 C5 4A 80 00
136: F6 20 23 C8 8E 80 80
139: F5 27 84 04 06 00
142: F6 0C 61 0F E4 06 03
145: F6 E2 24 07 10 22 38
148: F6 71 15 DA B5 51 1C
151: F6 04 7F 91 0C 04 78
154: F7 11 FE 0C 19 30 60 FF
157: F7 11 FE 0C 58 34 60 FF
163: F7 11 FE 0C 59 34 60 FF
166: F7 11 FE 0D 58 35 60 FF
169: F7 11 FE 0D 59 35 60 FF
172: F7 11 FE 0D D8 37 60 FF
175: F6 20 E3 EF EF 8E 08
178: F6 38 FB EF 8F 8F 8E
181: F6 30 F3 CF EF 0F 0C
184: F6 70 E5 FF F7 CE 1C
187: F5 01 C7 CF BF FF
190: F7 11 FF FB E7 C7 00 00
193: F5 01 C4 48 A0 C1
196: F7 11 06 0A 24 47 00 00
```

TECHNICAL DETAILS

RAM ROUTINE	SC	SIZE: 000

Stack Usage:
- 0 T: USED
- 1 Z: USED
- 2 Y: USED
- 3 X: USED
- 4 L: USED

Flag Usage:
- 04: NOT USED
- 05: NOT USED
- 06: NOT USED
- 07: NOT USED
- 08: NOT USED
- 09: NOT USED
- 10: USED
- 25: USED

Alpha Register Usage:
- 5 M: USED
- 6 N: NOT USED
- 7 O: NOT USED
- 8 P: NOT USED

Other Status Registers:
- 9 Q:
- 10 ⊦: NONE USED
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

Display Mode: ANY

Angular Mode: ANY

Unused Subroutine Levels: 3

ΣREG: NOT USED

Data Registers:
- R00:
- R06: NONE USED
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Global Labels Called:

Direct	Secondary
QR	NONE

Local Labels In This Routine:

00 to 14, 20 to 23

Execution Time:

Less than 3 seconds.

Peripherals Required:
None to run SC, but printer required to print char's

- Interruptible? YES
- Execute Anytime? YES
- Program File: N/A
- Bytes In RAM: 518
- Registers To Copy: N/A

Other Comments:

Use to load data registers with special characters, then RCL and ACSPEC later when they are needed.

VM - VIEW MANTISSA

VM will display, without printing, the full ten-digit mantissa of a number in the X register. The purpose is to allow the user to conveniently view all significant digits of the mantissa of a number without changing the calculator display mode and without affecting the X, Y, Z, T stack.

Example 1: Assume that the display mode is FIX 2 and the X register contains the number:
4.284453894 32
In FIX 2 display mode, this number will appear as:
4.28 32
Upon execution of VM, the user will see:
4.284453894
Note that pressing the back-arrow to clear the VIEW function reveals the X register to be unchanged, with the display once again appearing as:
4.28 32

COMPLETE INSTRUCTIONS FOR VM

VM may be used as any other single-argument function available in the HP-41. The only difference is that VM does not alter the X register, but does alter the display. With the number (argument) in question located in the X register, XEQ VM.

MORE EXAMPLES OF VM

Example 2: The following shows the contents of various registers both before and after executing VM. The numbers used are purely ficticious and any similarity between these and real, live numbers is purely coincidental. Assume FIX 4 display mode.

	BEFORE	AFTER
register T	1.020000000 00	1.020000000 00
Z	3.549288301 12	3.549288301 12
Y	3.141592653-51	3.141592653-51
X	2.997982818-03	2.997982818-03
LSTX	9.000000000 00	2.997982818-03
see...DISPLAY	0.0030	2.997982818

LINE BY LINE ANALYSIS OF VM

The routine will ultimately display the mantissa of the argument by using the hardwired VIEW function. All ten digits will be displayed by momentarily changing to FIX 9 display mode just prior to using VIEW. The user's original display conditions are saved and restored by RCLing and later STOing the contents of the d register which contains the status of all the flags, including those that pertain to the display mode. In order for the routine to end in the VIEW mode, the HP-41 must be in the VIEW mode when register d is first RCLed. This requires that a VIEW be performed before d is RCLed and before the mantissa is available for viewing. So flag 21, the printer usage flag, must be cleared to prevent this initial VIEW from printing.

01	LBL VM	Routine name.
02	CF 21	Keep printer from printing.
03	XROM MT	Call the PPC ROM mantissa Routine.
04	X<>M	Put the mantissa into the M register for safe keeping.
05	RDN	Roll stack to keep from losing the original contents of register T.
06	VIEW 0	Put the machine into VIEW mode.
07	RCL d	Save the original display & view status.
08	FIX 9	Set display to show all ten digits.
09	VIEW M	Display 10 digit mantissa.

VM

T E C H N I C A L   D E T A I L S

XROM: 10,26 — **VM** — SIZE: 000

Stack Usage:
- 0 T: UNCHANGED
- 1 Z: UNCHANGED
- 2 Y: UNCHANGED
- 3 X: UNCHANGED
- 4 L: X

Alpha Register Usage:
- 5 M: USED
- 6 N: USED
- 7 O: CLEARED
- 8 P: CLEARED

Other Status Registers: NONE USED
- 9 Q:
- 10 ⊢:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

ΣREG: UNCHANGED

Data Registers: NONE USED
- R00:
- R06:
- R07:
- R08:
- R09:
- R10:
- R11:
- R12:

Flag Usage:
- 04:
- 05:
- 06:
- 07:
- 08:
- 09:
- 10:
- 21: CLEARED
- 25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
MT	NONE

Local Labels In This Routine:
NONE

Execution Time: .9 seconds.

Peripherals Required: NONE

Interruptible? NO
Execute Anytime? NO
Program File: VM
Bytes In RAM: 26
Registers To Copy: 60

Other Comments:

10	STO d	Restore original display status.
11	RDN	Roll Stack to keep from losing T.
12	LASTX	Put argument back into X register.
13	RTN	Stop or return to calling routine.

CONTRIBUTORS HISTORY FOR VM

VM was written by Roger Hill (4940).

FURTHER ASSISTANCE ON VM

Call Keith Kendall (5425) at (801) 967-8080.
Call Roger Hill (4940) at (618) 656-8825.

Routine Listing For:	VM
01♦LBL "VM" 02 CF 21 03 XROM "MT" 04 X<> [05 RDN 06 VIEW] 07 RCL d	08 FIX 9 09 VIEW [10 STO d 11 RDN 12 LASTX 13 RTN

NOTES

VS - VERIFY SIZE

VS provides a "friendly" prompting feature for your programs that use data registers. VS gives a short TONE and prompts "RESIZE > = K" if the program requires SIZE k and the current SIZE is less than k.

Example 1: Suppose your program uses data registers 08 through 11. You can insert the following sequence of instructions at or near the top of the program to check that the SIZE is at least 12 and PROMPT if it isn't.

```
12
XROM VS
FC? C 25
PROMPT
```

COMPLETE INSTRUCTIONS FOR VS

To verify that the SIZE is at least k, call VS using the following sequence

```
k
XROM VS
FC? C 25
PROMPT
```

If the SIZE is less than k, there will be a short TONE followed by the prompt "RESIZE > = k".

The alpha register is not altered unless the SIZE is insufficient. This feature is essential in LB, (line 24) where VS is called while M and N are being used for temporary storage. The PROMPT is not included in VS, because resizing wipes out the return stack. If the PROMPT were in VS you would have to get out of ROM after resizing. This way you can just resize, and restart.

LINE BY LINE ANALYSIS OF VS

Line 61 stores X in L. Line 65 attempts to RCL data register x-1. If this succeeds, the RTN on line 68 is executed. Ohterwise the RESIZE prompt is composed, a short TONE (73) is executed, and control returns to the calling program.

CONTRIBUTORS HISTORY FOR VS

An early version of VS by Gary Tenzer (1816) appeared in (*PPC CALCULATOR JOURNAL*, V6N6P21c). Many intermediate versions were written, but this ROM version by Roger Hill (4940) has the useful feature that the alpha register is undisturbed unless resizing is required.

FURTHER ASSISTANCE ON VS

Call Keith Kendall (5425) at (801) 967-8080.

Call Roger Hill (4940) at (618) 656-8825.

Routine Listing For: VS

```
59◆LBL "VS"      69 "RESIZE>= "
60 SF 25         70 TONE 3
61 INT           71 R↑
62 RDN           72 RCL d
63 DSE T         73 FIX 0
64 ""            74 CF 29
65 RCL IND T     75 ARCL L
66 RDN           76 STO d
67 FS? 25        77 RDN
68 RTN           78 RTN
```

TECHNICAL DETAILS

XROM: 10,30 | VS | SIZE: 000

Stack Usage:
- 0 T: X-1 or d
- 1 Z: T
- 2 Y: Z
- 3 X: Y
- 4 L: X

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

REPLACED BY PROMPT IF INSUFFICIENT SIZE; OTHERWISE UNCHANGED.

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: UNCHANGED

Data Registers: NONE USED

R00:

R06:

R07:

R08:

R09:

R10:

R11:

R12:

Flag Usage: ONLY FLAG 25 USED

04:

05:

06:

07:

08:

09:

10:

25: USED: SET IN ROM AND CLEARED IN RAM.

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine: NONE

Execution Time: .5 seconds.

Peripherals Required: NONE

Interruptible? YES

Execute Anytime? NO

Program File: VM

Bytes In RAM: 44

Registers To Copy: 60

Other Comments:

VS

NOTES

XD - HEX TO DECIMAL

XD converts a two-digit hexadecimal number in the alpha register to a decimal number in the X register. This routine is used by the byte loader LB and can also be used as a utility subroutine providing the fastest means to convert a hexadecimal number (up to FF) to its decimal equivalent.

Example 1: Find the decimal equivalent of $C0_{16}$. Go into ALPHA mode and key in the string "C0". Switch back out of ALPHA mode and XEQ XD . This returns the decimal value 192 to X. This result can be verified by checking the decimal entry in row C, column 0 of the combined Hex Table.

Example 2: Find the decimal equivalent of $B010_{16}$. Since this is a four digit number and XD converts only two digit hexadecimal numbers, XD must be executed once for the first pair of digits and again for the second pair of digits.

DO:	SEE:	RESULT:
"B0"		Enter two hex digits
XEQ XD	176.	Decimal equivalent of $B0_{16}$
256		
*	45056.	Decimal equivalent of $B000_{16}$
"10"		Enter two hex digits
XEQ XD	16.	Decimal equivalent of 10_{16}
+	45072.	Decimal equivalent of $B010_{16}$.

COMPLETE INSTRUCTIONS FOR XD

Place a two digit (no more, no less) hexadecimal number "ab" in the alpha register and XEQ XD . The decimal value (16 a + b) of the hexadecimal number is place in the X register. The previous contents of X is copied into Y, Z, and T. Lastx contains the decimal equivalent of the first hex digit (16 a).

APPLICATION PROGRAM 1

The program "XD+" automates the procedure used in Example 2. It allows hexadecimal numbers of arbitrary length to be decoded into decimal, two hex digits at a time. The running total is kept in X. To see the result, just switch out of ALPHA mode when the prompt appears.

```
01  LBL "XD+"
02  CF 21
03  AON
04  .
05  LBL 00
06  "BYTE?"
07  AVIEW
08  CLA
09  STOP
10  256
11  *
12  XROM  XD
13  +
14  GTO  00
```

TECHNICAL DETAILS

XROM: 10,25 — **XD** — SIZE: 000

Stack Usage:

- 0 T: X
- 1 Z: X
- 2 Y: X
- 3 X: 0-255
- 4 L: 16*first digit

Alpha Register Usage:

- 5 M: USED
- 6 N: still clear
- 7 O: still clear
- 8 P: still clear

Other Status Registers: NONE USED

- 9 Q:
- 10 ⊦:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

ΣREG: UNCHANGED

Data Registers: NONE USED

R00:

R06:

R07:

R08:

R09:

R10:

R11:

R12:

Flag Usage: NONE USED

04:

05:

06:

07:

08:

09:

10:

25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 5

Global Labels Called:

Direct	Secondary
QR	NONE

Local Labels In This Routine:

NONE

Execution Time: 1.7 seconds.

Peripherals Required: NONE

Interruptible? YES

Execute Anytime? NO

Program File: LB

Bytes In RAM: 36 WITH END

Registers To Copy: 71

Other Comments:

XD

LINE BY LINE ANALYSIS OF XD

Line 241 appends the bytes 00 08 onto the two characters that were entered into ALPHA. Lines 242-244 recall the resulting four byte non-normalized number from the alpha register and call QR (Quotient/Remainder). In this instance the quotient is the numerical equivalent of the first character and the remainder is the equivalent of the second character. These numerical equivalents range in value from 30 to 39 (hex digits 0 to 9) and 41 to 46 (hex digits A to F). Lines 245-253 calculate the decimal values of each hexadecimal digit using the clever formula INT [(x-29) * 0.9]. Lines 254-257 multiply the first digit by 16 and add it to the second digit.

CONTRIBUTORS HISTORY FOR XD

XD was written by Roger Hill (4940) as a fast conversion routine to add hex input capability to LB.

FURTHER ASSISTANCE ON XD

Call Roger Hill (4940) at (618) 656-8825.

Call Keith Kendall (5425) at (801) 967-8080.

Routine Listing For: **XD**

```
240♦LBL "XD"
241 "├♦å"
242 RCL [
243  E2
244 XROM "QR"
245 29
246 ST- Z
247 -
248 .9
249 ST* Z
250 *
251 INT
252 X<>Y
253 INT
254 16
255 *
256 +
257 END
```

NOTES

XE - XROM ENTRY

XE allows you to enter at any line in a ROM program, not just at the ROM's global labels. Two main benefits are a) you can for your own use "get at" routines in the ROM's which otherwise you could not, and b) you can omit leading parts of routines which contain undesirable lines near their beginning. The latter permits adapting programs originally written only for manual use so that they operate properly as subroutines. **XE** uses synthetic lines, and the user must set up the ROM entry address for **XE** by one of several short synthetic-program methods given below. **XE** uses and clears the alpha register, but preserves the stack.

Example 1: Your program must display or print the names of days and months, and you find them all set up for you in the CLNDR routine in your STRD PAC, complete with an XEQ IND X for calling them (line 147 in STRD 1B). There is no global label there, but **XE** comes to the rescue, saving you almost 140 bytes.

1) Review CLNDR; with 1 to 19 in X, entering CLNDR at line 147 XEQ IND X, exit will be at line 149 RTN, with the name of a day or month in the alpha register. By Flag 21 you control line 148 AVIEW: print the result en route, stop and see it (R/S to continue), or run through and assemble your output later.

2) Write your program using any register such as R00 to store the ROM entry address. For example, try DOM (day or month) as follows:

```
01  LBL  "DOM"
02  ARCL  00
03  XEQ  "XE"
04  END
```

3) Set up the ROM entry address in R00:

DO:	SEE:	RESULT:
RTN		Clears b return stack for ASTO
GTO"CLNDR"		Access the ROM
GTO.147		GTO line 147 in the ROM
PRGM	147 XEQ IND X	Check ROM entry line
PRGM (off)		
RCL b	0.0000000-07	Program pointer in X
CLA		Clears alpha for ASTO
STO M		Program pointer into M
ASTO 00	⊠ ⊠	Store it in R00

Do the above RCL b and STO M by assigned keys (see **MK**) or synthetic barcode.

4) Put 1 to 19 in X, XEQ"DOM" - it works!

5) Write the rest of your program.

6) Repeat step 3) whenever you set up for running DOM, or store R00 on a data card, but be sure to have the STRD 1B in the port where it was in step 3), since the address in R00 depends on the port.

Example 2: You can save RAM space and at the same time make your program self-contained (no manual setup before running it) by changing line 02 in DOM into a synthetic text statement, thus eliminating the need for register R00. (For an example, see routine **TN** lines 124, 125.) One way to create the text line is:

1) GTO"CLNDR", GTO.147, RCL b

2) Decode the NNN in X, getting decimal bytes (194, 147) for the two right-most bytes. (STRD 1B in port 3) See **NH** , **CD** , or **2D** .

3) Create the two-character text line using **LB** and the numbers (242,194,147), placing it adjacent to line 02 in DOM and deleting the ARCL 00 line. The check version of DOM then becomes

```
01  LBL "DOM"
02  ""
03  XROM XE
04  END
```

Be forewarned about the printer and synthetic texts. The ROM entry address in display formats (X and alpha) is shown in step 3 of Example 1, but the two-character text prints as nothing between the quotes, in line 02. For a complete discussion of this printer behavior, see *PPC Calculator Journal*, V7N6P20c.

COMPLETE INSTRUCTIONS FOR XE

XE starts with the M register containing the address of the line in a ROM at which you want to enter as a subroutine call from your program. (The address must be in the two right-most bytes of M, in the format of the program pointer as it would be obtained by a manual RCL b, STO M after manual GTO/SST to the desired line in the ROM.) **XE** uses and clears the alpha register, but returns the stack as it was. In nested subroutines, **XE** in effect uses up one level, so using **XE** you can have five-deep subroutines instead of six, including the ones pending at this point in your program, the ROM you will call, and its subroutine depth. **XE** clears flag 14, but uses no data registers.

1) Review the ROM program you want to use. You want an entry point (no label necessary) which, starting with alpha and Flag 14 cleared does your job. If you want it to return control to your program without stops like a proper subroutine, you must be fortunate enough to find a well-placed RTN or END. Find a STOP or PROMPT and you've had it. An AVIEW or VIEW is O.K. if you can avoid setting Flag 21 or if you'll accept printing the item. (See Example 3)

2) Decide which way to handle the ROM entry address, then use A, B, or C below.

A. Address in program text line (preferred)

3) Manually GTO the ROM program line where you want execution to start, and do a RCL b.

4) Decode the two right-most bytes of X, getting decimal numbers Byte 1 and Byte 0. (See **NH**, **CD** or 2D)

5) With **LB** create text line in your program using decimal bytes (242, Byte 1, Byte 0).

6) Place the line XROM **XE** in your program immediately after the above text line.

7) To run your program, just be sure that the ROM you are calling is in the same port as it was in step 3) above, with the PPC ROM placed in any port.

Refer to HEX Table on page 16.

B. Address in data register

3) Place the lines ARCL nn, XROM **XE** in that order in your program.

4) To run your program, set up the register Rnn. First, manually RTN, GTO the desired ROM entry line, and RCL b. Then manually or in your program do CLA, STO M, ASTO nn. Your program runs and reruns as long as you don't disturb Rnn or put the ROM into a different port.

C. Address on data card

5) After completing B, store the Rnn contents on a magnetic card, along with other data your program may need.

6) Now to run your program just load the data from the card, be sure the ROM is in the right port, and begin.

MORE EXAMPLES OF XE

Example 3: You need a triangle subroutine, and SSS in the MATH PAC computes what you want, but it includes unwanted PROMPTs for data and sets Flag 21 so the many AVIEWs cause an inordinate amount of stops if no printer is there. All is fixed if SSS is entered at line 06 LBL 05, having pre-placed sides 1, 2, and 3 in registers 00,02, and 04, and in Z,Y, and X respectively. Choosing method A, which puts the address in a text line:

1) Manually GTO"SSS", and in PRGM, SST to see 06 LBL 05, with MATH 1A in port 3.

2) Poff (PRGM mode off), USER, RCL b.

3) Using **NH**, decode the X register contents, getting the two decimal bytes (201,174).

4) GTO.. and then use **LB** to create the synthetic two-character text line using decimal bytes (242, 201,174), looking like 01 ᵀ ⊠ ⊠ in the HP-41C display.

5) Write a check program around the synthetic text, which becomes line 08 below. (Note that the 174 resulted in skipping 14 spaces!)

```
01  LBL "MAIN"        07  STO 04
02  25                08  "              "
03  STO 00            09  XEQ XE
04  35                10  "OK"
05  STO 02            11  AVIEW
06  45                12  END
```

6) After you XEQ"MAIN" you see some flashing results and it stops, displaying "OK". The triangle's angles are in R01, R03, and R05.

Example 4: To write a program to compute the cumulative detection probability for a scanning sensor with Gaussian statistics, as it completes successive scans, a subroutine is needed for calculating the probability that a detection threshold is exceeded on each scan, given by

$$Q(X) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} e^{-\frac{s^2}{2}}\, ds$$

where x is a function of target signal amplitude and threshold setting. Q(x) will be computed repetitively for the sequence of values of x computed by the main program. A routine for Q(x) is in the STAT PAC, but only for manual inputs. However, there is no need to copy it out into RAM and massage it; use **XE** and save work and RAM space. This example is more involved than the others, but gets results from a less friendly routine. Assume that STAT 1B is in port 3. A look at ΣNORMD reveals that it runs in two parts. The front part loads constants and sets flags, and the LBL e part computes Q(x) when done; the stack and the alpha register are used; registers R03,R04,R05,R06,R07, R08 must have data from the front part of ΣNORMD: data gets stored into R01,R02,R09; Flags 00,01,02 are cleared; exit, after an AVIEW, is at line 163 RTN. So far, O.K.

The front part, executed at ΣNORMD calls LBL"*b" which sets Flag 27 (USER mode) and Flag 21 (print enable) plus other things not needed, and at the end of the front part, after loading data into most of the registers through R16, it ends at 30 STOP. Not so good.

The main program can be set up to use **XE** twice. First, avoid the *b call by entering below it at line 03 .2316419 where loading data into R03 to R08 and R11 to R16 begins. The program screeches to a halt at 30 STOP, after a BEEP triumphantly announces the ΣNORMD display -- no way to fix this, but it will only happen once, and the easiest way out is to plan to press R/S. That just lets the STAT PAC do its thing on a harmless short calculation of its f(x), the normal distribution function, and without stopping it returns to the main program. An immediate CLD will eliminate the f(x) display.

Since the only data needed are in R03 to R08, the rest of the registers are available for use after the first **XE** call, but it requires initially SIZE ≥ 017, and R01,R02,R09 are scratch memory.

The other **XE** call is to compute Q(x), entering after LBL E at line 50 STO 01.

The following formulas define the problem, with a simplified expression for x. P_m is the cumulative probability of detection.

$$P_o = 0$$

$$P_n = P_{n-1} + (1-P_{n-1})\, Q(x) \quad \text{for } n = 1,2,3\cdots$$

$$x = 3-.3n$$

$$\text{Stop when } P_n \geqslant .9$$

The final program will print as follows, including the two **XE** calls and their preceding text lines, treated below.

Lines 02 and 03 loads data.
Lines 13 and 14 computes Q(x).

APPLICATION PROGRAM FOR: **XE**

```
01♦LBL "POD"          14 XROM "XE"
 02 "x̄"                15 1
03 XROM "XE"           16 RCL 00
 04 CLD                17 -
 05 FIX 3              18 *
 06 0                  19 ST+ 00
 07 STO 00             20 .9
 08 3                  21 RCL 00
 09 STO 10             22 X>Y?
                       23 STOP
10♦LBL 00              24 .3
 11 RCL 10             25 ST- 10
 12 VIEW 00            26 GTO 00
 13 ""                 27 END
```

The text for each of the synthetic text lines is determined as in examples 2) and 3): with STAT 1B in port 3.

DO:	SEE: (FIX 9)
For line 02:	
GTO"ΣNORMD"	
PRGM	01 LBL T ΣNORMD
SST,SST	03 .2316419
Poff,Rcl b	0.000000001$
Decode X getting two decimal bytes (206,2)	
For line 13	
GOT"ΣNORMD",STO E	
PRGM	49 LBL E
SST	50 STO 01
Poff,RCL b	0.000000001%
Decode X getting two decimal bytes (206,156)	

Creating these two synthetic text lines together, use **LB** with decimal byte sequence (242,206,2,242,206,156). Then key in the rest of the program.

To run the program, XEQ"POD", and after the BEEP fanfare for ΣNORMD, push R/S.

During running, the display alternately shows Q(x) (with Q=) and P_m (just the number): not bad after all.

Example 5: ROM exploration: Charles Close's (3878) article on "Romping thru ROM" opened an extremely fruitful topic (see reference), using a modified predecessor of **XE** and a byte jumper to read RAM-compatible code or ROM microcode (assembly language) out of a ROM. Using his method, copy **XE** into RAM, label it XE+ and insert lines FC? 14, STOP after the CLA line. Read his article before doing anything, but with a chosen address in M one can XEQ XE+ and the second SST sets down in the ROM ready for a byte-jump.

However, with the PPC ROM there are now faster ways. See the write ups on **Sb** and **Ab** for this and their other uses.

FURTHER DISCUSSION OF XE

The PPC ROM routine **Rb** can be used to determine the port number of the PPC ROM. This allows one to enter PPC ROM routines without regard to what port the ROM is in. See the **Rb** write up for details.

The **Sb** and **Ab** routines offer a quick but somewhat limited method to enter any ROM. No RTN is possible unless you set it up synthetically. See the **Sb** and **Ab** write ups.

LINE BY LINE ANALYSIS OF XE

Lines 120-121 shift the subroutine return stack one place to the left (two bytes) in registers a and b. Register b now contains, counting from the right end, the program pointer in bytes 0 and 1, then the first subroutine return address back to XEQ 14 in bytes 2 and 3, and so forth with preceding returns. That return address in bytes 2 and 3 will never be used, since the main purpose of the rest of this routine is to slice out bytes 2 and 3 and replace them with the desired ROM entry address. Then when the RTN is executed later, control goes to the ROM, and exit from the ROM will go back to the point from which **XE** was called, etc.

Line 122 shifts the ROM entry address, in the M register, into bytes 2 and 3, lining it up with its destination in the b register. Note that the M content was simply the program pointer, read when at the ROM entry line. This works because the formats in the return address file and program pointer are the same for ROM addresses, although not for RAM return addresses.

Lines 123-131 park X in N, then b is read and switched into N, reconstituting the original stack. SF 14 prevented execution of lines 129 and 131 this time, but then is cleared preparing for exit on the next pass.

Lines 132-140 do the shift-slice-and-patch operations to assemble the new b content, as described above. The stack remains in order.

Lines 141-142 put X into N, and the new b is then installed in the b register. It got its program pointer when line 126 was executed, so now the next line to be executed will be 127.

Line 127 puts X back in the stack. This is the subtle part of **XE**. To preserve the stack X has to be re-placed, after the new b is installed, by this line which didn't look special the first time through it. Like a Sci-Fi story preserving casualty in spite of a backward time jump.

Lines 128-131 now clear alpha, and the RTN sends control to the desired ROM entry, from which return reverts to the program or subroutine chain which originally called **XE**.

REFERENCES FOR XE

V7N5P55 "Direct Addressing of ROM Routines" by William C. Wickes (3735). Re-reading this is guaranteed to brighten your day. *PPC CALCULATOR JOURNAL*, V7N7P16 lists Wickes' ROM routine as XRE.
V8N1P14 "Romping Thru ROM" by Charles Close (3878) opened a door to future HP-41C capabilities. (By XRTN he meant XRE which evolved to **XE**.)

CONTRIBUTORS HISTORY FOR XE

William C. Wickes (3735), frustrated by the inflexibility of the MATH PAC in subroutine usage, devised the original concept and wrote the ROM program. Les Matson (5608) revised it permitting a call from up to a fourth-deep subroutine. Roger Hill (4940) wrote the final version, which is a bit shorter and facilitates addressing by a short text line in the calling program.

FURTHER ASSISTANCE ON XE

Call Les Matson (5608) at W- (617) 258-1764 or
H- (617) 235-7955.

Call Roger Hill (4940) at (618) 656-8825.

Routine Listing For:	XE
119◆LBL "XE"	131 RTN
120 XEQ 14	132 "⊦***"
	133 X<> [
121◆LBL 14	134 X<> \
122 "⊦**"	135 X<> [
123 STO \	136 "⊦**"
124 RDN	137 STO \
125 SF 14	138 X<>]
126 RCL b	139 X<> \
127 X<> \	140 "⊦**"
128 FC? 14	141 X<> \
129 CLA	142 STO b
130 FC?C 14	

TECHNICAL DETAILS

XROM: 10,19	XE	SIZE: 000

Stack Usage: 0 T: 1 Z: ALL UNCHANGED 2 Y: 3 X: 4 L:	Flag Usage: ONLY FLAG 14 USED 04: 05: 06: 07: 08:
Alpha Register Usage: 5 M: 6 N: ALL CLEARED 7 O: 8 P:	09: 10: 14: CLEARED 25:
Other Status Registers: 9 Q: NOT USED 10 ⊢: NOT USED 11 a: NOT USED 12 b: MODIFIED 13 c: NOT USED 14 d: NOT USED 15 e: NOT USED	Display Mode: UNCHANGED Angular Mode: UNCHANGED Unused Subroutine Levels: 5
ΣREG: UNCHANGED Data Registers: NONE USED R00: R06: R07: R08: R09: R10: R11: R12:	Global Labels Called: Direct: NONE Secondary: NONE Local Labels In This Routine: 14
Execution Time: .8 seconds.	
Peripherals Required: A ROM	
Interruptible? YES Execute Anytime? NO Program File: ML Bytes In RAM: 58 Registers To Copy: 64	Other Comments:

NOTES

XL - XROM INPUTS FOR LB

XL is a support program for LB and the MK family of MK , 1K , and -K . XL accepts two XROM inputs and converts the XROM numbers into decimal HEX table numbers. If the XROM number is represented as XROM AA, BB as seen in the HP-41 display, the two inputs for XL would be A and B. The two decimal bytes that XL produces may be loaded into memory as an XROM call or they may be used to assign the XROM instruction to a key.

Example 1: Assign XROM 20,57 to key 15.

First convert 20,57 into the corresponding Instruction bytes. FIX 2.

DO:	SEE:	RESULT:
20, ENTER	20.00	A Input for XL
57	57	B Input for XL
XEQ XL	165.00	A Byte decimal value
X⇄Y	57.00	B Byte decimal value
15	15	Key input
XEQ 1K	0.00	XROM 20,57 is assigned to key 15, the LN key.

Example 2: Ed is trying to explain to Bob that the HP-41 memory uses eight bit Byte instructions as arranged in the HEX table. He explains that two bytes may be interpreted by the HP-41 operating system according to their order. To illustrate how this works Bob asks, "Can you place two bytes into memory that will print the the alpha register, PRA, even if the printer isn't available?" That's easy Ed replied, PRA is XROM 29,08. Ed uses his PPC ROM and keys 29, ENTER, 8, XEQ XL. The bytes he loads into memory are 16 and 72 using LB. (See LB for details of how this is done.) Single stepping through memory Ed shows Bob that he has placed an arbitrary XROM number into memory. He plugs in a printer and the XROM 29,08 instruction becomes PRA.

COMPLETE INSTRUCTIONS FOR XL

The XL routine is intended for keyboard use in support of LB. It is similar to BL and FL in this regard, except that XL may be called as a subroutine. Two inputs are required, A and B. The outputs are the decimal values of the XROM instructions (HEX A0 through A7). The stack conditions before and after execution are shown below:

T: T		→	T: Z
Z: Z			Z: Z
Y: A	XEQ XL		Y: Byte "B"
X: B	produces		X: Byte "A"
L: L			L: 256

The 256 is a by-product of the QR routine used by XL. QR uses Register 0.

FURTHER DISCUSSION OF XL

The HP-41C/CV operating system is designed to address up to 64 K of ROM. This addressing capability is divided into 16 4K ROM's. Half of these 16 ROMs are "hard addressed" and half of these are port addressed. If we number these ROM's 0 to 16, the ROM's would be identified as shown in table 1. Each ROM has an identification number stored in it. Each ROM may contain 64 global labels. The HP-41 operating system processes this information to produce an XROM number AA, BB which identifies a specific function in the ROM

TABLE 1. HP-41 ROM USAGE

No.	Use	No.	Use
0	Internal ROM	8	Lower 4K of Port 1
1	Internal ROM	9	Upper 4K of Port 1
2	Internal ROM	10	Lower 4K of Port 2
3	Spare?, Internal?	11	Upper 4K of Port 2
4	Service Module	12	Lower 4K of Port 3
5	Future ROM?	13	Upper 4K of Port 3
6	82143A Printer	14	Lower 4K of Port 4
7	Future ROM?	15	Upper 4K of Port 4

The numbers AA and BB have the following structure:

AA - 0 to 31 (0 is not usable)
BB - 0 to 63 (0 is the ROM ID "function").

CAUTION: XL does not check inputs for valid range and will produce mathematically correct results, but meaningless instruction bytes if the input range is not restricted as shown above.

The HEX table--refer to the table on page 16 or the pocket plastic card--has 8 instructions dedicated to XROM instructions. Each XROM number has 64 functions (maximum) which means that a given XROM instruction may use 256 post fix bytes or four groups of 64. XROM 00,00 would be HEX A0,00 or 160,00 decimal. XROM 31,63 would be HEX A7,63 or 167,63 decimal. XROM numbers may be visualized by referring to the table below.

TABLE 2. XROM NUMBERS

XROM	BYTES	XROM	BYTES
00,00	160,00	28,00	167,00
00,63	160,63	28,63	167,63
01,00	160,64	29,00	167,64
01,63	160,127	29,63	167,127
02,00	160,128	30,00	167,128
02,63	160,191	30,63	167,191
03,00	160,192	31,00	167,192
03,63	160,255	31,63	167,63
04,00	161,00	32,00	INVALID
04,63	161,63	etc.	etc.
etc.	etc.		

The pocket "HP-41 Combined HEX/Decimal Byte Table" identifies these XROM number ranges with XR 0 thru 3 for 160, XR 4 thru 7 for 161, etc.

APPLICATION PROGRAM 1 FOR XL

The concept of Example 1 may be programmed to automatically assign any XROM instruction to a key with the short program below. Input is: A, ENTER, B, ENTER, Key No., XEQ XR to K.

```
01♦LBL "XR TO K"
02 X<> T
03 RDN
04 XROM "XL"
05 X<>Y
06 R↑
07 X<> T
08 XROM "1K"
09 RTN
```

Assign XL to key 31: 20, ENTER, 57, ENTER, 31, XEQ XR to K. Press the shift key and see ⊤ XL . Oops. How do we clear the assignment? Easy! XEQ CK .

XL

Routine Listing For:	XL
32♦LBL "XL" 33 X<>Y 34 640 35 + 36 64 37 *	38 + 39 256 40 XROM "QR" 41 X<>Y 42 RTN

LINE BY LINE ANALYSIS OF XL

XL utilizes a clever algorithm that produces the first byte value 160 to 167 when the "A" input ranges from 0 to 31 and a corresponding second byte value from 0 to 255 when the "B" input ranges from 0 to 63. This is done by computing N in lines 33 thru 38 using:

$$N = 64A + B + 40960$$

This single number, N, is divided by 256. The quotient is the first byte, the remainder is the second byte. This is done using the QR routine in lines 39 and 40. The X exchange Y at line 41 places the first byte in the X register. The algorithm, like many algorithms in the computing field are difficult to "explain". They often are the result of the creative insight of the programmer.

REFERENCES FOR XL

See PPC CJ, V8N2P41b for the most recent ROM PROGRESS reference to XL . Also see PPCJ, V7N2P31b.

CONTRIBUTORS HISTORY FOR

A fancy program along the lines suggested in the second reference was envisioned for the ROM. As bytes became premium entities, this idea was abandoned. Roger Hill (4940) came to the rescue with a short, fast, non-prompting version that is usable as a subroutine.

FINAL REMARKS FOR XL

XL , like FL and BL are "programmers aid" type routines that are a class of functions that must be in future PPC's. Programs that tell the user the labels used and their frequency, programs or functions that aid in debugging or optimizing subroutine usage, all belong in this class of programs. XL , FL , and BL were "last minute" support programs. Only XL approaches being an efficient program.

FURTHER ASSISTANCE ON XL

Call Richard Nelson (1) at (714) 754-6226 P.M.
Call Roger Hill (4940) at (618) 656-8825.

HP-41C COMBINED HEX/DECIMAL BYTE

	0	1	2	3	4	5	6	7	8	9
8	DEG IND 00 128 ◆	RAD IND 01 129 ×	GRAD IND 02 130 x̄	ENTER↑ IND 03 131 ←	STOP IND 04 132 α	RTN IND 05 133 β	BEEP IND 06 134 Γ	CLA IND 07 135 ↓	ASHF IND 08 136 Δ	PSE IND 09 137 σ
9	RCL IND 16 144 θ	STO IND 17 145 Ω	ST+ IND 18 146 δ	ST− IND 19 147 Ä	ST* IND 20 148 ä	ST/ IND 21 149 Ä	ISG IND 22 150 ä	DSE IND 23 151 Ö	VIEW IND 24 152 ö	ΣREG IND 25 153 Ü
→ A	XR 0-3 IND 32 160	XR 4-7 IND 33 161 !	XR8-11 IND 34 162 "	X12-15 IND 35 163 #	X16-19 IND 36 164 $	X20-23 IND 37 165 %	X24-27 IND 38 166 &	X28-31 IND 39 167 '	SF IND 40 168 (	CF IND 41 169)
B	SPARE IND 48 176 0	GTO 00 IND 49 177 1	GTO 01 IND 50 178 2	GTO 02 IND 51 179 3	GTO 03 IND 52 180 4	GTO 04 IND 53 181 5	GTO 05 IND 54 182 6	GTO 06 IND 55 183 7	GTO 07 IND 56 184 8	GTO 08 IND 57 185 9

TECHNICAL DETAILS

XROM: 20,57 **XL** SIZE: 000

Stack Usage:
- 0 T: USED
- 1 Z: USED
- 2 Y: USED
- 3 X: USED
- 4 L: USED (256)

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O:
- 8 P:

NONE USED BY ROUTINE*

Other Status Registers:
- 9 Q:
- 10 ⊢:
- 11 a:
- 12 b:
- 13 c:
- 14 d:
- 15 e:

NONE USED BY ROUTINE

ΣREG: NOT USED

Data Registers:
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

NONE USED BY ROUTINE

Flag Usage: NONE
04:
05:
06:
07:
08:
09:
10:
25:

Display Mode: N/A

Angular Mode: N/A

Unused Subroutine Levels: 4

Global Labels Called:

Direct	Secondary
QR	NONE

Local Labels In This Routine:
NONE

Execution Time: Approximately 1 second.

Peripherals Required: NONE

Interruptible? YES
Execute Anytime? NO
Program File: SM
Bytes In RAM: 24
Registers To Copy: 26

Other Comments:
* Reg. 0 is used by QR.

Σ?

Σ? - ΣREG FINDER

If you have a printer, finding the location of the statistical register block is easy--just XEQ "PRFLAGS". If you don't have a printer, or if you want to generate and use the register number in a program, you're out of luck--that is, until now. Through the wonders of synthetic programming we can manipulate the c register to reveal the secrets it holds.

Σ? will give you the number of the first register of the statistical block. This routine executes in about 2 seconds and it can be called from a program.

Example 1: After MASTER CLEAR, XEQ **Σ?** returns a value of 11. This means that the statistical register block is as follows: $R_{11} = \Sigma x$, $R_{12} = \Sigma x^2$, $R_{13} = \Sigma y$, $R_{14} = \Sigma y^2$, $R_{15} = \Sigma xy$, $R_{16} = n$.

COMPLETE INSTRUCTIONS FOR Σ?

XEQ **Σ?** to place the decimal number of the ΣREG block in X. The previous contents of X are duplicated in Y, Z, and T. The curtain location is placed in L; adding this to the result in X will give the absolute address of the ΣREG block.

A typical application of **Σ?** arises in the case in which a program processes data entered by the user via the Σ+ key. The normal procedure is for the program to set ΣREG to a fixed location before user data entry. **Σ?** allows a ΣREG location to be used which has been previously set by the user. The program can execute **Σ?** to find out where the data is. The data can then be accessed through INDirect STO, RCL, and x<> instructions.

MORE EXAMPLES OF Σ?

Example 2: To compute the absolute location of the ΣREG block after master clear use XEQ **Σ?**, LASTX, +, yielding 250.

Example 3: If you use **ΣC** to move the curtain for subroutine use, you may encounter a situation where an error has interrupted execution and you don't know where the curtain is. The standard recovery procedure is to execute **S?** , subtract the original SIZE, then execute **CU** to restore the original curtain location. But in this case there is an alternate procedure that doesn't require that you know the original SIZE. Just XEQ **Σ?** . If the result is positive then the curtain is in its original position and the number in X indicates the ΣREG location that was set before the first call to **ΣC**. If the result is negative then the ΣREG pointer is being used to hold the previous curtain pointer (and vice versa). For instance, -14 indicates that **ΣC** has been used to raise the curtain 14 registers. To restore the original curtain location simply XEQ **ΣC**.

LINE BY LINE ANALYSIS OF Σ?

23	LBL **Σ?**	
24	CLA	clear alpha to ensure that no garbage is present
25	XROM **C?**	return curtain address to x-register, leave Σ-register pointer in 5th and 6th bytes of N

TECHNICAL DETAILS

XROM: 10,14 — **Σ?** — SIZE: 000

Stack Usage:
- 0 T: X
- 1 Z: X
- 2 Y: X
- 3 X: ΣREG location
- 4 L: curtain location

Flag Usage: MANY USED BUT ALL RESTORED
04:
05:
06:
07:
08:
09:
10:
25:

Alpha Register Usage:
- 5 M:
- 6 N:
- 7 O: ALL CLEARED
- 8 P:

Other Status Registers:
- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: NOT USED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 5

ΣREG: UNCHANGED

Data Registers: NONE USED
R00:
R06:
R07:
R08:
R09:
R10:
R11:
R12:

Global Labels Called:

Direct	Secondary
C?	NONE

Local Labels In This Routine:
NONE

Execution Time: 2.2 seconds.

Peripherals Required: NONE

Interruptible?	YES
Execute Anytime?	YES
Program File:	**ML**
Bytes In RAM:	18
Registers To Copy:	64

Other Comments:

```
26  RCL N
27  XEQ 14        convert hex Σ-register location to
                  decimal absolute address
28  CLA           clear alpha of garbage
29  X<>Y
30  -             subtract absolute address of Σ-regis-
                  ters from curtain address to obtain
                  relative address
31  RTN
```

See explanation of **C?** program to provide further insight into line 27.

CONTRIBUTORS HISTORY FOR **Σ?**

The idea of a ΣREG finder appears to have originated in the *PPC CALCULATOR JOURNAL*, V6N8P17, with Craig Pearce (311). Synthetic versions (V7N5P12a and V7N7P15d) were then written by Steve Wandzura (4635) and Valentin Albillo (4747). **Σ?** was written by Keith Jarett (4360) as part of the integrated **S?** / **Σ?** /**ΣC**? package.

FURTHER ASSISTANCE ON **Σ?**

Call Clifford Stern (4516) at (213) 748-0706.

Call Keith Jarett (4360) at (213) 374-2583.

Routine Listing For:	**Σ?**
23♦LBL "Σ?" 24 CLA 25 XROM "C?" 26 RCL \\ 27 XEQ 14 28 CLA 29 X<>Y 30 - 31 RTN 48♦LBL 14 49 STO [50 "├♦♦A" 51 X<> [52 X<> d 53 CF 01 54 CF 02 55 CF 04 56 CF 07 57 FS?C 10 58 SF 07	59 FS?C 11 60 SF 09 61 FS?C 12 62 SF 10 63 FS?C 13 64 SF 11 65 FS?C 14 66 SF 13 67 FS?C 15 68 SF 14 69 FS?C 16 70 SF 15 71 X<> d 72 E38 73 / 74 INT 75 DEC 76 RTN

NOTES

ΣC - ΣREG CURTAIN EXCHANGE

This very fast routine interchanges the pointers in status register c to R00 and to the statistical block of registers. **ΣC** can repeatedly raise and lower the curtain by n registers, if the calls are preceded by a setup ΣREG n command.

BACKGROUND FOR **ΣC**

See the appendix on Curtain Moving.

Example 1: The sequence

```
      :
ΣREG  08
XROM  ΣC
XEQ   "SUB"
XROM  ΣC
      :
```

raises the curtain by 8 registers, calls on a subroutine "SUB", then lowers the curtain to its former position (provided that "SUB" did not tamper with the contents of status register c--via a ΣREG instruction, for example). Three inputs and three outputs may be carried in the stack, since **ΣC** preserves X, Y, and Z.

COMPLETE INSTRUCTIONS FOR **ΣC**

ΣC is used in a sequence as shown in Example 1. **ΣC** leaves X, Y, and Z unchanged. ALPHA is cleared and a temporary c register is left in T.

ΣC has one advantage over the **UD** / **HD** combination for raising and lowering the curtain to protect register contents from the processing of a called subroutine--it doesn't require that the called subroutine leave R00 alone. On the other hand, unlike the **HD** / **UD** combination, **ΣC** does require that the two pointers in status register c are returned to the calling routine with the same values they had upon entry to the called routine. For this reason, **ΣC** is not the appropriate tool for more than one level of curtain manipulation in a chain of calls each of which requires curtain movement.

The use of **ΣC** to raise and lower the curtain entails an approximately ½ second time penalty over the **HD** / **UD** combination. Note: **ΣC** is not interruptible if the printer is attached and (**E?** mod 8) ≤ 3. This is because the printer may find flag 55 clear at line 167, in which case it would set flags 55 and 21. Flag 21 is at that time part of the .END. pointer, and it may need to be clear.
If (E? mod 8) < 4, increase or decrease SIZE by 4 to make **ΣC** interruptible. This procedure also works for **HD** and **CU**.

MORE EXAMPLES OF **ΣC**

Example 2: Suppose you wish to use **SV** to find a root to your function "FXR", which happens to use data registers 00 through 07. This conflicts with **SV**'s use of registers 06 through 09. Since "FXR" uses R00, **HD** / **UD** cannot be used. The solution is to use **ΣC**. (This example assumes that "FXR" does not change ΣREG.) Insert the following steps at the beginning and end of the "FXR" function program:

```
LBL "FXR"
ΣREG 10
XROM ΣC
  :
body of "FXR"
  :
XROM ΣC
RTN
```

Then just alpha store "FXR" in R_{06} and XEQ **SV** to find the root.

LINE BY LINE ANALYSIS OF **ΣC**

Regard the initial contents of status register c as the following 14 hex digits:

$$s_1s_2\ s_30\ 01\ 69\ z_1z_2\ z_3e_1\ e_2e_3$$

where: $s_1s_2s_3$ = the abs. 3 hex-digit address of the first Σ-register
$z_1z_2z_3$ = the abs. 3 hex-digit address of R00
$e_1e_2e_3$ = the abs. 3 hex-digit address of reg. containing .END.

The **ΣC** routine interchanges $s_1s_2s_3$ with $z_1z_2z_3$. When **ΣC** finishes, the new contents of status register c is $z_1z_2\ z_30\ 01\ 69\ s_1s_2\ s_3e_1\ e_2e_3$.

Getting $z_1z_2z_3$ into the $s_1s_2s_3$-slots is easy, via a ΣREG 00 (see line 177). The hard part of getting $s_1s_2s_3$ into the $z_1z_2z_3$ slots is replacing the 'z_3e_1' byte by 's_3e_1'. This replacement entails two phases: isolate the 's_30' byte; replace the 2nd hex digit by 'e_1'.

(1) By line 165, alpha register N contains 00 00 00 00 s_30 00 00 and stack register X contains $z_1z_2\ z_3e_1\ e_2e_3$ 00 00 00 00.

(2) In order to append hex digit 'e_1' to hex digit 's_3', we position byte 's_30' in flag register d so that it corresponds to flags 32 through 39, and position hex digit 'e_1' as the 2nd hex digit in register N wherein the 1st bit is set (in line 167 to ensure that the contents of N are interpreted as numeric) and the last byte is '00' (to ensure that only the 2nd hex digit--the integer part of the number--is relevant in the 'SCI IND N' instruction at line 172). When line 172 is executed,

reg. d contains 00 00 00 00 s_30 00 00
reg. N contains ue_1 e_2e_3 00 00 00 00 00
(u ≥ 8 to guarantee numeric interpretation)

Thus 'SCI IND N' means SCI e_1 (and, in fact, e_1 would be either 0 or 1). In this way, flags 32 through 39 become the needed byte 's_3e_1'. A simple check of stack operations shows that upon exit from **ΣC**, the

TECHNICAL DETAILS

XROM: 10,21	ΣC	SIZE: Σ? + 6

Stack Usage:

- 0 T: temporary c
- 1 Z: UNCHANGED
- 2 Y: UNCHANGED
- 3 X: UNCHANGED
- 4 L: USED

Alpha Register Usage:

- 5 M:
- 6 N: ALL CLEARED
- 7 O:
- 8 P:

Other Status Registers:

- 9 Q: NOT USED
- 10 ⊦: NOT USED
- 11 a: NOT USED
- 12 b: NOT USED
- 13 c: ALTERED
- 14 d: USED BUT RESTORED
- 15 e: NOT USED

ΣREG: PREVIOUS CURTAIN

Data Registers: NONE USED

R00:

R06:

R07:

R08:

R09:

R10:

R11:

R12:

Flag Usage:

04:
05: FLAG REGISTER USED BUT RESTORED
06:
07:
08:
09:
10:

25:

Display Mode: UNCHANGED

Angular Mode: UNCHANGED

Unused Subroutine Levels: 6

Global Labels Called:

Direct	Secondary
NONE	NONE

Local Labels In This Routine:

NONE

Execution Time: 1.6 seconds.

Peripherals Required: NONE

Interruptible? ONLY IF PRINTER NOT ATTACHED*

Execute Anytime? NO

Program File: ML

Bytes In RAM: 102 WITH END

Registers To Copy: 64

Other Comments:
*If printer is attached there is a 50% chance that ΣC cannot be interrupted. If you have trouble, change SIZE by 4. See text for remedy.

contents of registers X, Y, and Z are what they were upon entry to ΣC ; the contents of register T, however, has changed.

REFERENCES FOR ΣC

See *PPC CALCULATOR JOURNAL*, V7N5P45 for an introductory discussion.

CONTRIBUTORS HISTORY FOR ΣC

ΣC was conceived and written by Keith Jarett (4360). An improved ΣC (V8N2P2) was written by Clifford Stern (4516) several weeks after the ROM was assembled. That version is fully interruptible with the printer attached.

FURTHER ASSISTANCE ON ΣC

Call Harry Bertuccelli (3994) at (213) 846-6390.
Call Keith Jarett (4360) at (213) 374-2583.

Routine Listing For: ΣC

```
154♦LBL "ΣC"
155 CLA
156 RCL c
157 STO ]
158 STO [
159 "⊦A"
160 CLX
161 STO \
162 "⊦B"
163 STO [
164 "⊦CD"
165 X<> ]
166 X<> d
167 SF 08
168 X<> d
169 X<> \
170 "⊦E"
171 X<> d
172 SCI IND \
173 X<> d
174 STO ]
175 RDN
176 RCL c
177 ΣREG 00
178 X<> c
179 STO [
180 "⊦F"
181 RDN
182 RCL \
183 "⊦GHI"
184 STO L
185 RDN
186 RCL c
187 STO [
188 X<> ]
189 "⊦J"
190 STO [
191 "⊦K"
192 X<> L
193 STO [
194 "⊦L"
195 X<> \
196 X<> c
197 RDN
198 CLA
199 END
```

NOTES

APPENDIX M - CURTAIN MOVING

INTRODUCTION:

The HP-41 operating system permits the user to vary the number of data registers via the built-in function SIZE. When SIZE is executed, a pointer in status register c is adjusted to change the location of data register R00, so that data register Rn, where n = Size - 1, is the last available memory register. The term curtain, originally used by Bill Wickes (see PPC CJ, V6N8P27d), has become the accepted name for this movable partition at R00 between data and program. When the curtain is moved via SIZE, both data and program are shifted by the operating system. The curtain-moving routines in the PPC ROM (**CU** , **CX** , **HD** , **UD** , and **ΣC**) shift neither data nor program; they simply alter the pointer used by the operating system to locate data registers; Rn is located at absolute address z + n, where z is the value of the pointer to R00 found in status register c.

For example, suppose SIZE = 41. Figure A depicts the structure of user memory. Absolute addresses shown assume the full RAM complement of 320 registers. The symbols inside the rectangles (depicting data registers) represent the actual content of those registers.

(HEX)	absolute address (DECIMAL)		user address	
1FF	511	d_{40}	R40	
1FE	510	d_{39}	R39	
1FD	509	d_{38}	R38	Data Registers
	⋮		⋮	
1D9	473	d_{02}	R02	
1D8	472	d_{01}	R01	
1D7	471	d_{00}	R00	"curtain"
1D6	470		-	
1D5	469	User	-	Program Area
1D4	468	Programs	-	
1D3	467		-	
	⋮		⋮	

FIGURE A.

Figure B contrasts the register contents after the curtain is raised two different ways. The curtain can be raised 10 registers either via resizing to SIZE = 31 or via the PPC ROM curtain mover **CU** . Both ways of moving the curtain change the size to 31, but the **CU** technique does not change the contents of the registers. In particular d_{31} through d_{40} are retained and the registers are effectively renumbered so that the contents of R_k is d_{k+10}, the former contents of R_{k+10}. This has a very important consequence. The former contents of registers R00 through R09 are now held in suspended animation in that part of memory now regarded by the operating system as user program area. This data held below the curtain is safe from being disturbed by STO or RCL until **CU**

absolute address	SIZE = 31	user address	10 XEQ CU	
511	d_{30}	R30	d_{40}	
510	d_{29}	R29	d39	
509	d_{28}	R28	d38	Data Registers
⋮		⋮		
483	d_{02}	R02	d_{12}	
482	d_{01}	R01	d_{11}	
481	d_{00}	R00	d_{10}	New curtain
480		-	d_{09}	
479	User	-	d_{08}	
478	Programs	-	⋮	Program Area
⋮				
471		-	d_{00}	
470	←--- previous curtain location	-		
469		-	User	
468		-	Programs	
467		-		
⋮		⋮		

FIGURE B.

is used again (with an input of -10) to reverse the register renumbering.

The most important application of curtain moving routines is to deal with conflicting register usage between a program and a subroutine or program that it calls (via XEQ). Normally when register usage conflicts, you have to rewrite one program or the other to eliminate the conflict. This is tedious, but even worse is the case in which the subroutine is arbitrary and its register usage cannot be determined ahead of time.

With curtain moving routines one is free to ignore subroutine use of data registers. If your main program requires that R00-R07 remain intact while a subroutine is called, it can simply raise the curtain 8 registers before calling the subroutine and lower it again when execution returns to the main program. As far as the subroutine is concerned, it has free use of all available registers, but the data required by the main program is tucked safely away below the curtain. Rootfinders and similar programs can use this register renumbering technique to insure full compatibility with any user-supplied subroutine. In accordance with the no free lunch principle, the total number of data registers used is the same with or without curtain moving.

Another application of curtain moving is to allow PPC ROM routines to be run with a smaller SIZE. Since most PPC ROM routines that use data registers start with R06, the SIZE required to run a ROM routine is normally six larger than the number of registers used. As long as you won't be accessing R00-R05 you can SIZE to six less than the normal requirement and lower the curtain 6 registers into program memory. (-6 XEQ CU). Be sure to put the curtain back (6 XEQ CU) before using R00 through R05 or you'll wipe out part of your top program.

A brief overview of the routines CU, CX, HD, and ΣC should prove helpful in selecting which is most appropriate to a particular application.

CU (CUrtain Up) and CX (Curtain to absolute X)

These are the most powerful of the five (curtain raising or lowering, in any sequence) and impose no constraints on subsequent processing. However, they're also the slowest, so that other choices are preferable where their constraints can be accepted.

HD (Hide Data) and UD (Uncover Data)

This complementary pair of routines for raising and lowering the curtain (respectively) are the fastest means for the type of curtain manipulation required to call within a computing loop a routine which otherwise would destroy data needed by the calling routine. The main requirement is that, after raising the curtain, the routine called does not change the contents of the new R00, which is used by HD to hold the last 5 bytes of the previous contents of status register c so that UD can lower the curtain very quickly to its former position.

ΣC (interchange pointers to Σ registers and Curtain)

This very fast routine interchanges pointers to R00 and to the statistical block of registers. In fact, this code is used by HD. Since ΣC doesn't use a data register, it lays no register-avoiding requirement as does HD upon a routine called after raising the curtain. However, such a called routine should not execute a ΣREG-instruction, directly or indirectly (via a call on a subroutine that does). This is because the ΣREG pointer is being used to temporarily hold the previous curtain location.

With the curtain control routines shown here, not only can one program renumber the registers before it calls another program, but this second program can do a second renumbering before calling a third program, etc., creating a multi-level data "stack" if needed. The critical sequence of steps to be embedded in any program to enable it to correctly call another program ("W" in the following examples) can be any one of the four options shown below.

Various curtain control techniques to protect R_{00} - R_{k-1} from subroutine "W"

PROPER CALLING SEQUENCE

SEQ.:	1	2	3	4
	k	k	ΣREG k	k
	XROM CU	XROM HD	XROM ΣC	XROM HD
	XEQ "W"	XEQ "W"	XEQ "W"	XEQ "W"
	-k	-k	XEQ ΣC	XROM UD
	XROM CU	XROM CU		

RUN TIME-- EXC. "W":	roughly 7 sec.	roughly 5.5 sec.	2.2 sec.	1.9 sec.

CONSTRAINTS:

1. Standard: Don't PACK with the curtain up. Don't branch backwards to a numerical label in the top program.
2. Same as 1. plus: ΣREG setting is changed.
3. Same as 2. plus: "W" can't contain any ΣREG instructions or use ΣC or HD internally.
4. Same as 2. plus: "W" can't use data register 00 (R'_{00}).

The first two methods place no restrictions on "W". The second method moves the summation register block. The fourth method is the fastest, but requires that "W" not disturb register R'_{00}. (The new R_{00}). The third method is probably the best choice for root finder programs and other single-level renumbering applications where R'_{00} cannot be reserved.

The "standard constraints" need some discussion. If you PACK with the curtain up, any null bytes in data below the curtain will be removed. Not only does this effectively destroy the data, but it also repositions the program below it, so that when the curtain is restored some of the program will be in data registers. To partially protect against this problem, PACK before raising the curtain over data registers. The HP-41 will then think that the top program is already packed and will not attempt to repack it.

The second constraint is due to the label search logic. If the first program (at the top of the user program area) includes an uncompiled backward branch, then whenever data registers are below the curtain, execution of this branch will entail a search for the label in these data registers. Should a hex code corresponding to the label be found there, execution would continue by interpreting subsequent bytes in these data registers as program. Labels 00 through 08 are most likely to cause trouble. To ensure that label searching is confined to the actual program area, place an END at the top of program memory (and PACK it).

APPLICATIONS OF CURTAIN MOVING

The most important problem addressed by these routines is some variant of the following. You are designing a rather complicated program which uses some data registers, and you can simplify the task by using existing subroutines to do some portions of the job. However, when these subroutines were written, the programmers weren't thinking about this application, and these subroutines are using data registers you want to use. (This problem can become especially acute when the number of data registers required is problem dependent, and thus is established during execution.) Instead of scattering your use of data registers around those used by a subroutine, you are free to ignore subroutine use of data registers by simply moving the curtain beyond the last register you need before calling a subroutine that uses data registers; when the subroutine returns control to the calling program, you can return the curtain to its former position. Such curtain manipulations can be catenated in a nested calling sequence.

Example 1: Suppose the main program needs to retain information in registers R00 to R04, and calls subroutine XX which needs to retain information in R01 and R02 and calls, in turn, subroutine YY, which uses registers R00 and R01. Assume that the call on subroutine XX is within a computing loop. This makes routines HD and UD the preferred curtain movers since XX doesn't use register R00. On the other hand, our assumption that YY does use register R00 (so that XX wouldn't be able to reinstate curtain position via UD) rules against the use of HD by XX prior to calling YY. If we assume that YY has no need to execute a ΣREG (implicit in the preceding description that YY only used registers R00 and R01), then XX can use ΣC to raise the curtain before calling YY. The following program segments show how HD, UD, and ΣC facilitate the described use of subroutines XX and YY:

Main Program	Subroutine
.	LBL "XX"
.	ΣREG 03
.	.
5	.
XROM HD	.
XEQ "XX"	XROM ΣC
XROM UD	XEQ "YY"
.	XROM ΣC
.	.
.	.
END	.
	END

The PPC ROM routine CU could have been used for all of the curtain manipulation in the previous example. Like ΣC it does not employ any data registers. Furthermore, it leaves the pointer to the statistical registers undisturbed, and in some contexts this could prove preferrable. However, ΣC is substantially faster, and UD, where practical, is by far the fastest way to lower the curtain to its previous location. The HD - UD combination to raise and subsequently lower the curtain is faster than two applications of ΣC.

See the Hanoi Tower Puzzle Generalized, covered elsewhere in this manual, for an elaborate example of curtain manipulation.

END

NOTES

APPENDIX - N BARCODES OF ROM ROUTINES

BA

PROGRAM REGISTERS NEEDED: 49

ROW 1 (1 : 4)

ROW 2 (4 : 12)

ROW 3 (12 : 21)

ROW 4 (22 : 27)

ROW 5 (28 : 33)

ROW 6 (33 : 38)

ROW 7 (38 : 43)

ROW 8 (43 : 48)

ROW 9 (48 : 52)

ROW 10 (52 : 59)

ROW 11 (59 : 61)

ROW 12 (61 : 61)

ROW 13 (61 : 69)

ROW 14 (70 : 76)

ROW 15 (77 : 84)

ROW 16 (85 : 93)

ROW 17 (93 : 99)

ROW 18 (100 : 105)

ROW 19 (105 : 113)

ROW 20 (114 : 121)

ROW 21 (121 : 128)

ROW 22 (128 : 130)

ROW 23 (131 : 138)

ROW 24 (139 : 147)

ROW 25 (148 : 157)

ROW 26 (157 : 164)

BD

PROGRAM REGISTERS NEEDED: 53

ROW 1 (1 : 5)

ROW 2 (6 : 12)

ROW 3 (12 : 21)

ROW 4 (21 : 31)

ROW 5 (32 : 38)

ROW 6 (38 : 38)

ROW 7 (38 : 49)

ROW 8 (49 : 55)

ROW 9 (55 : 62)

ROW 10 (63 : 71)

ROW 11 (71 : 78)

ROW 12 (79 : 85)

ROW 13 (85 : 92)

ROW 14 (93 : 98)

ROW 15 (99 : 108)

ROW 16 (109 : 116)

ROW 17 (117 : 123)

ROW 18 (123 : 132)

ROW 19 (133 : 141)

ROW 20 (141 : 149)

ROW 21 (150 : 156)

ROW 22 (157 : 160)

ROW 23 (160 : 165)

ROW 24 (166 : 170)

ROW 25 (170 : 179)

ROW 26 (179 : 187)

ROW 27 (187 : 194)

ROW 28 (195 : 202)

ROW 29 (202 : 206)

BL

PROGRAM REGISTERS NEEDED: 46

ROW 1 (1 : 7)

ROW 2 (8 : 14)

ROW 3 (15 : 21)

ROW 4 (21 : 30)

ROW 5 (31 : 40)

ROW 6 (41 : 49)

ROW 7 (50 : 57)
ROW 8 (58 : 64)
ROW 9 (65 : 70)
ROW 10 (70 : 76)
ROW 11 (76 : 83)
ROW 12 (83 : 90)
ROW 13 (90 : 93)
ROW 14 (93 : 100)
ROW 15 (100 : 108)
ROW 16 (108 : 116)
ROW 17 (116 : 125)
ROW 18 (125 : 132)
ROW 19 (133 : 137)
ROW 20 (137 : 141)
ROW 21 (142 : 148)
ROW 22 (148 : 155)
ROW 23 (155 : 160)
ROW 24 (161 : 167)
ROW 25 (167 : 168)

CA

PROGRAM REGISTERS NEEDED: 38

ROW 1 (1 : 5)
ROW 2 (5 : 12)
ROW 3 (13 : 21)
ROW 4 (21 : 27)
ROW 5 (27 : 36)
ROW 6 (37 : 44)
ROW 7 (45 : 54)
ROW 8 (54 : 61)
ROW 9 (62 : 68)
ROW 10 (69 : 79)
ROW 11 (80 : 87)
ROW 12 (88 : 96)
ROW 13 (96 : 106)
ROW 14 (106 : 112)
ROW 15 (113 : 119)

ROW 16 (120 : 126)
ROW 17 (126 : 133)
ROW 18 (134 : 141)
ROW 19 (142 : 151)
ROW 20 (152 : 160)
ROW 21 (160 : 160)

CV

PROGRAM REGISTERS NEEDED: 42

ROW 1 (1 : 5)
ROW 2 (6 : 15)
ROW 3 (16 : 26)
ROW 4 (26 : 34)
ROW 5 (35 : 43)
ROW 6 (44 : 51)
ROW 7 (52 : 61)
ROW 8 (62 : 70)
ROW 9 (71 : 81)
ROW 10 (81 : 91)
ROW 11 (91 : 100)
ROW 12 (100 : 110)
ROW 13 (111 : 118)
ROW 14 (118 : 127)
ROW 15 (128 : 135)
ROW 16 (135 : 141)
ROW 17 (142 : 152)
ROW 18 (152 : 162)
ROW 19 (162 : 168)
ROW 20 (168 : 178)
ROW 21 (179 : 185)
ROW 22 (186 : 192)
ROW 23 (192 : 194)

FI

PROGRAM REGISTERS NEEDED: 47

ROW 1 (1 : 6)
ROW 2 (7 : 16)
ROW 3 (17 : 23)

ROW 4 (24 : 29)

ROW 5 (30 : 37)

ROW 6 (37 : 45)

ROW 7 (45 : 53)

ROW 8 (54 : 65)

ROW 9 (65 : 74)

ROW 10 (75 : 86)

ROW 11 (87 : 96)

ROW 12 (97 : 107)

ROW 13 (107 : 118)

ROW 14 (119 : 127)

ROW 15 (128 : 139)

ROW 16 (140 : 150)

ROW 17 (151 : 161)

ROW 18 (162 : 174)

ROW 19 (175 : 186)

ROW 20 (187 : 195)

ROW 21 (196 : 205)

ROW 22 (206 : 215)

ROW 23 (215 : 225)

ROW 24 (226 : 234)

ROW 25 (234 : 243)

FR

PROGRAM REGISTERS NEEDED: 36

ROW 1 (1 : 5)

ROW 2 (6 : 13)

ROW 3 (14 : 21)

ROW 4 (22 : 29)

ROW 5 (29 : 36)

ROW 6 (36 : 45)

ROW 7 (46 : 50)

ROW 8 (51 : 61)

ROW 9 (62 : 74)

ROW 10 (75 : 84)

ROW 11 (85 : 94)

ROW 12 (95 : 99)

ROW 13 (100 : 110)

ROW 14 (111 : 120)

ROW 15 (121 : 127)

ROW 16 (127 : 134)

ROW 17 (134 : 142)

ROW 18 (143 : 148)

ROW 19 (148 : 152)

ROW 20 (153 : 154)

IF

PROGRAM REGISTERS NEEDED: 60

ROW 1 (1 : 6)

ROW 2 (7 : 14)

ROW 3 (15 : 21)

ROW 4 (21 : 28)

ROW 5 (29 : 35)

ROW 6 (36 : 41)

ROW 7 (42 : 46)

ROW 8 (47 : 52)

ROW 9 (52 : 60)

ROW 10 (61 : 68)

ROW 11 (69 : 76)

ROW 12 (77 : 82)

ROW 13 (82 : 91)

ROW 14 (91 : 96)

ROW 15 (97 : 100)

ROW 16 (101 : 106)

ROW 17 (107 : 115)

ROW 18 (115 : 122)

ROW 19 (122 : 128)

ROW 20 (129 : 133)

ROW 21 (134 : 142)

ROW 22 (142 : 145)

ROW 23 (145 : 151)

ROW 24 (152 : 157)

ROW 25 (158 : 162)

ROW 26 (163 : 169)

ROW 27 (170 : 175)

ROW 28 (176 : 181)

ROW 29 (181 : 186)

ROW 30 (186 : 191)

ROW 31 (191 : 196)

ROW 32 (196 : 205)

ROW 33 (206 : 206)

IG

PROGRAM REGISTERS NEEDED: 43

ROW 1 (1 : 6)

ROW 2 (7 : 15)

ROW 3 (15 : 25)

ROW 4 (26 : 37)

ROW 5 (37 : 47)

ROW 6 (48 : 56)

ROW 7 (57 : 67)

ROW 8 (67 : 75)

ROW 9 (75 : 82)

ROW 10 (83 : 91)

ROW 11 (91 : 97)

ROW 12 (98 : 107)

ROW 13 (107 : 116)

ROW 14 (116 : 125)

ROW 15 (125 : 130)

ROW 16 (130 : 138)

ROW 17 (139 : 146)

ROW 18 (146 : 155)

ROW 19 (156 : 164)

ROW 20 (165 : 173)

ROW 21 (174 : 179)

ROW 22 (179 : 187)

ROW 23 (188 : 197)

ROW 24 (197 : 197)

LB

PROGRAM REGISTERS NEEDED: 71

ROW 1 (1 : 5)

ROW 2 (5 : 7)

ROW 3 (7 : 11)

ROW 4 (11 : 17)

ROW 5 (18 : 24)

ROW 6 (25 : 33)

ROW 7 (34 : 41)

ROW 8 (41 : 49)

ROW 9 (49 : 57)

ROW 10 (57 : 64)

ROW 11 (65 : 74)

ROW 12 (74 : 84)

ROW 13 (84 : 91)

ROW 14 (91 : 95)

ROW 15 (96 : 102)

ROW 16 (102 : 108)

ROW 17 (108 : 116)

ROW 18 (117 : 127)

ROW 19 (127 : 135)

ROW 20 (135 : 143)

ROW 21 (144 : 147)

ROW 22 (147 : 152)

ROW 23 (153 : 158)

ROW 24 (158 : 161)

ROW 25 (161 : 163)

ROW 26 (163 : 169)

ROW 27 (169 : 178)

ROW 28 (178 : 187)

ROW 29 (188 : 195)

ROW 30 (196 : 203)

ROW 31 (203 : 207)

ROW 32 (207 : 214)

ROW 33 (215 : 222)

ROW 34 (223 : 232)

ROW 35 (232 : 239)

ROW 36 (240 : 243)

ROW 37 (243 : 250)

ROW 38 (251 : 257)

LF

PROGRAM REGISTERS NEEDED: 59

ROW 1 (1 : 5)
ROW 2 (6 : 10)
ROW 3 (10 : 17)
ROW 4 (18 : 24)
ROW 5 (24 : 31)
ROW 6 (31 : 38)
ROW 7 (39 : 44)
ROW 8 (44 : 52)
ROW 9 (52 : 59)
ROW 10 (60 : 65)
ROW 11 (65 : 72)
ROW 12 (72 : 78)
ROW 13 (78 : 83)
ROW 14 (84 : 91)
ROW 15 (92 : 99)
ROW 16 (100 : 105)
ROW 17 (105 : 112)
ROW 18 (112 : 118)
ROW 19 (118 : 123)
ROW 20 (123 : 129)
ROW 21 (129 : 136)
ROW 22 (136 : 143)
ROW 23 (143 : 150)
ROW 24 (151 : 156)
ROW 25 (156 : 164)
ROW 26 (164 : 170)
ROW 27 (171 : 176)
ROW 28 (177 : 185)
ROW 29 (185 : 191)
ROW 30 (192 : 197)
ROW 31 (198 : 204)
ROW 32 (205 : 209)

M2

PROGRAM REGISTERS NEEDED: 61

ROW 1 (1 : 5)
ROW 2 (6 : 10)
ROW 3 (11 : 18)
ROW 4 (19 : 26)
ROW 5 (27 : 31)
ROW 6 (31 : 35)
ROW 7 (36 : 43)
ROW 8 (44 : 53)
ROW 9 (54 : 60)
ROW 10 (61 : 66)
ROW 11 (67 : 74)
ROW 12 (75 : 85)
ROW 13 (86 : 92)
ROW 14 (93 : 97)
ROW 15 (97 : 103)
ROW 16 (103 : 109)
ROW 17 (110 : 118)
ROW 18 (119 : 126)
ROW 19 (126 : 132)
ROW 20 (133 : 141)
ROW 21 (141 : 150)
ROW 22 (151 : 155)
ROW 23 (155 : 162)
ROW 24 (163 : 172)
ROW 25 (173 : 182)
ROW 26 (183 : 191)
ROW 27 (191 : 197)
ROW 28 (198 : 206)
ROW 29 (206 : 212)
ROW 30 (213 : 218)
ROW 31 (219 : 229)
ROW 32 (230 : 235)
ROW 33 (235 : 239)

MK

PROGRAM REGISTERS NEEDED: 61

ROW 1 (1 : 5)
ROW 2 (5 : 15)
ROW 3 (15 : 21)
ROW 4 (22 : 24)
ROW 5 (24 : 30)
ROW 6 (30 : 33)
ROW 7 (33 : 39)
ROW 8 (39 : 42)
ROW 9 (42 : 52)
ROW 10 (53 : 61)
ROW 11 (62 : 71)
ROW 12 (71 : 80)
ROW 13 (80 : 88)
ROW 14 (89 : 96)
ROW 15 (99 : 105)
ROW 16 (106 : 112)
ROW 17 (112 : 118)
ROW 18 (118 : 125)
ROW 19 (125 : 130)
ROW 20 (131 : 138)
ROW 21 (138 : 144)
ROW 22 (145 : 152)
ROW 23 (153 : 158)
ROW 24 (158 : 159)
ROW 25 (160 : 163)
ROW 26 (164 : 172)
ROW 27 (172 : 174)
ROW 28 (174 : 178)
ROW 29 (178 : 184)
ROW 30 (184 : 187)
ROW 31 (188 : 195)
ROW 32 (195 : 202)
ROW 33 (203 : 209)

ML

PROGRAM REGISTERS NEEDED: 64

ROW 1 (1 : 4)
ROW 2 (5 : 12)
ROW 3 (12 : 17)
ROW 4 (17 : 21)
ROW 5 (22 : 27)
ROW 6 (27 : 33)
ROW 7 (33 : 41)
ROW 8 (42 : 47)
ROW 9 (47 : 52)
ROW 10 (53 : 59)
ROW 11 (59 : 65)
ROW 12 (66 : 72)
ROW 13 (72 : 78)
ROW 14 (78 : 80)
ROW 15 (80 : 86)
ROW 16 (86 : 91)
ROW 17 (91 : 98)
ROW 18 (98 : 103)
ROW 19 (103 : 109)
ROW 20 (110 : 117)
ROW 21 (118 : 122)
ROW 22 (122 : 128)
ROW 23 (129 : 134)
ROW 24 (135 : 140)
ROW 25 (140 : 143)
ROW 26 (144 : 150)
ROW 27 (150 : 155)
ROW 28 (156 : 162)
ROW 29 (162 : 167)
ROW 30 (167 : 173)
ROW 31 (173 : 180)
ROW 32 (180 : 185)
ROW 33 (186 : 191)
ROW 34 (191 : 197)
ROW 35 (198 : 199)

MP

PROGRAM REGISTERS NEEDED: 86

ROW 1 (1 : 4)
ROW 2 (4 : 11)
ROW 3 (11 : 16)
ROW 4 (16 : 22)
ROW 5 (23 : 31)
ROW 6 (32 : 38)
ROW 7 (39 : 44)
ROW 8 (44 : 47)
ROW 9 (48 : 52)
ROW 10 (52 : 58)
ROW 11 (59 : 69)
ROW 12 (69 : 78)
ROW 13 (78 : 85)
ROW 14 (86 : 93)
ROW 15 (94 : 100)
ROW 16 (101 : 111)
ROW 17 (112 : 121)
ROW 18 (122 : 133)
ROW 19 (133 : 142)
ROW 20 (142 : 151)
ROW 21 (152 : 162)
ROW 22 (163 : 172)
ROW 23 (172 : 180)
ROW 24 (181 : 188)
ROW 25 (189 : 197)
ROW 26 (198 : 203)
ROW 27 (204 : 212)
ROW 28 (212 : 219)
ROW 29 (220 : 228)
ROW 30 (229 : 237)
ROW 31 (238 : 245)
ROW 32 (246 : 255)
ROW 33 (255 : 263)
ROW 34 (263 : 269)
ROW 35 (270 : 278)
ROW 36 (279 : 285)
ROW 37 (285 : 292)
ROW 38 (292 : 296)
ROW 39 (296 : 306)
ROW 40 (307 : 314)
ROW 41 (314 : 319)
ROW 42 (319 : 323)
ROW 43 (323 : 329)
ROW 44 (329 : 334)
ROW 45 (334 : 340)
ROW 46 (340 : 345)

NH

PROGRAM REGISTERS NEEDED: 33

ROW 1 (1 : 6)
ROW 2 (6 : 10)
ROW 3 (11 : 16)
ROW 4 (16 : 23)
ROW 5 (23 : 30)
ROW 6 (31 : 37)
ROW 7 (38 : 44)
ROW 8 (44 : 51)
ROW 9 (51 : 58)
ROW 10 (58 : 65)
ROW 11 (66 : 72)
ROW 12 (72 : 78)
ROW 13 (79 : 85)
ROW 14 (86 : 92)
ROW 15 (92 : 98)
ROW 16 (99 : 105)
ROW 17 (105 : 111)
ROW 18 (112 : 114)

NS

PROGRAM REGISTERS NEEDED: 16

ROW 1 (1 : 6)
ROW 2 (7 : 15)

ROW 3 (15 : 20)
ROW 4 (21 : 24)
ROW 5 (24 : 29)
ROW 6 (30 : 36)
ROW 7 (37 : 42)
ROW 8 (42 : 48)
ROW 9 (48 : 51)

S1

PROGRAM REGISTERS NEEDED: 47

ROW 1 (1 : 8)
ROW 2 (9 : 19)
ROW 3 (20 : 29)
ROW 4 (30 : 35)
ROW 5 (36 : 45)
ROW 6 (46 : 55)
ROW 7 (55 : 63)
ROW 8 (64 : 73)
ROW 9 (73 : 80)
ROW 10 (81 : 89)
ROW 11 (90 : 100)
ROW 12 (100 : 108)
ROW 13 (109 : 118)
ROW 14 (119 : 128)
ROW 15 (129 : 138)
ROW 16 (139 : 145)
ROW 17 (145 : 153)
ROW 18 (154 : 161)
ROW 19 (161 : 169)
ROW 20 (169 : 177)
ROW 21 (178 : 185)
ROW 22 (186 : 193)
ROW 23 (194 : 201)
ROW 24 (202 : 209)
ROW 25 (209 : 216)

SM

PROGRAM REGISTERS NEEDED: 26

ROW 1 (1 : 4)
ROW 2 (4 : 11)
ROW 3 (12 : 18)
ROW 4 (18 : 27)
ROW 5 (28 : 32)
ROW 6 (33 : 40)
ROW 7 (40 : 45)
ROW 8 (46 : 50)
ROW 9 (50 : 54)
ROW 10 (54 : 57)
ROW 11 (58 : 64)
ROW 12 (65 : 73)
ROW 13 (73 : 79)
ROW 14 (80 : 87)

SR

PROGRAM REGISTERS NEEDED: 40

ROW 1 (1 : 6)
ROW 2 (6 : 12)
ROW 3 (12 : 18)
ROW 4 (18 : 24)
ROW 5 (24 : 26)
ROW 6 (27 : 34)
ROW 7 (35 : 39)
ROW 8 (39 : 45)
ROW 9 (45 : 51)
ROW 10 (52 : 57)
ROW 11 (57 : 64)
ROW 12 (65 : 70)
ROW 13 (71 : 76)
ROW 14 (77 : 80)
ROW 15 (81 : 87)
ROW 16 (87 : 92)
ROW 17 (93 : 99)
ROW 18 (99 : 107)

ROW 19 (108 : 115)

ROW 20 (115 : 121)

ROW 21 (122 : 130)

ROW 22 (131 : 131)

VK

PROGRAM REGISTERS NEEDED: 63

ROW 1 (1 : 5)

ROW 2 (5 : 11)

ROW 3 (11 : 16)

ROW 4 (16 : 20)

ROW 5 (20 : 27)

ROW 6 (28 : 34)

ROW 7 (35 : 42)

ROW 8 (43 : 52)

ROW 9 (53 : 63)

ROW 10 (63 : 71)

ROW 11 (72 : 80)

ROW 12 (80 : 85)

ROW 13 (85 : 92)

ROW 14 (92 : 98)

ROW 15 (98 : 106)

ROW 16 (107 : 114)

ROW 17 (115 : 118)

ROW 18 (119 : 123)

ROW 19 (124 : 130)

ROW 20 (131 : 138)

ROW 21 (139 : 149)

ROW 22 (150 : 156)

ROW 23 (157 : 160)

ROW 24 (161 : 165)

ROW 25 (165 : 170)

ROW 26 (170 : 175)

ROW 27 (175 : 181)

ROW 28 (182 : 188)

ROW 29 (189 : 194)

ROW 30 (195 : 198)

ROW 31 (199 : 206)

ROW 32 (206 : 213)

ROW 33 (214 : 222)

ROW 34 (223 : 227)

VM

PROGRAM REGISTERS NEEDED: 60

ROW 1 (1 : 5)

ROW 2 (6 : 13)

ROW 3 (14 : 19)

ROW 4 (20 : 27)

ROW 5 (27 : 32)

ROW 6 (33 : 40)

ROW 7 (40 : 46)

ROW 8 (46 : 50)

ROW 9 (50 : 55)

ROW 10 (56 : 62)

ROW 11 (63 : 69)

ROW 12 (69 : 73)

ROW 13 (74 : 79)

ROW 14 (79 : 84)

ROW 15 (85 : 94)

ROW 16 (95 : 102)

ROW 17 (103 : 108)

ROW 18 (108 : 116)

ROW 19 (116 : 121)

ROW 20 (121 : 126)

ROW 21 (127 : 131)

ROW 22 (131 : 136)

ROW 23 (136 : 143)

ROW 24 (143 : 153)

ROW 25 (153 : 161)

ROW 26 (161 : 169)

ROW 27 (169 : 174)

ROW 28 (174 : 179)

ROW 29 (179 : 183)

ROW 30 (183 : 189)

ROW 31 (190 : 197)

ROW 32 (198 : 205)

ROW 33 (205 : 205)

NOTES

NOTES

NOTES

APPENDIX O – BARCODES OF APPLICATIONS PROGRAMS

MAXMIN

PROGRAM REGISTERS NEEDED: 31

ROW 1 (1 : 3)
ROW 2 (3 : 5)
ROW 3 (5 : 8)
ROW 4 (8 : 11)
ROW 5 (11 : 15)
ROW 6 (15 : 16)
ROW 7 (16 : 21)
ROW 8 (21 : 24)
ROW 9 (25 : 29)
ROW 10 (29 : 36)
ROW 11 (37 : 43)
ROW 12 (43 : 49)
ROW 13 (50 : 59)
ROW 14 (59 : 65)
ROW 15 (65 : 70)
ROW 16 (70 : 75)
ROW 17 (75 : 77)

MPP/HPP

PROGRAM REGISTERS NEEDED: 22

ROW 1 (1 : 4)
ROW 2 (4 : 7)
ROW 3 (7 : 12)
ROW 4 (12 : 17)
ROW 5 (17 : 24)
ROW 6 (25 : 31)
ROW 7 (31 : 34)
ROW 8 (34 : 37)
ROW 9 (37 : 41)
ROW 10 (42 : 46)
ROW 11 (46 : 52)
ROW 12 (53 : 56)

SMP/SHP

PROGRAM REGISTERS NEEDED: 12

ROW 1 (1 : 5)
ROW 2 (6 : 14)
ROW 3 (15 : 23)
ROW 4 (23 : 28)
ROW 5 (28 : 37)
ROW 6 (38 : 45)
ROW 7 (45 : 45)

ACV

PROGRAM REGISTERS NEEDED: 18

ROW 1 (1 : 6)
ROW 2 (6 : 15)
ROW 3 (16 : 24)
ROW 4 (25 : 31)
ROW 5 (32 : 35)
ROW 6 (35 : 41)
ROW 7 (41 : 45)
ROW 8 (46 : 50)
ROW 9 (50 : 56)
ROW 10 (56 : 57)

CPP

PROGRAM REGISTERS NEEDED: 10

ROW 1 (1 : 3)
ROW 2 (4 : 14)
ROW 3 (15 : 25)
ROW 4 (26 : 36)
ROW 5 (36 : 41)

ACP

PROGRAM REGISTERS NEEDED: 15

ROW 1 (1 : 4)
ROW 2 (4 : 10)
ROW 3 (11 : 14)
ROW 4 (15 : 20)
ROW 5 (20 : 27)

ROW 6 (27 : 34)

ROW 7 (35 : 41)

ROW 8 (42 : 47)

MPT/HPT

PROGRAM REGISTERS NEEDED: 23

ROW 1 (1 : 4)

ROW 2 (4 : 8)

ROW 3 (9 : 15)

ROW 4 (16 : 24)

ROW 5 (25 : 30)

ROW 6 (30 : 34)

ROW 7 (34 : 39)

ROW 8 (40 : 42)

ROW 9 (42 : 48)

ROW 10 (48 : 49)

ROW 11 (50 : 53)

ROW 12 (54 : 60)

MIO

PROGRAM REGISTERS NEEDED: 19

ROW 1 (1 : 3)

ROW 2 (3 : 7)

ROW 3 (7 : 10)

ROW 4 (11 : 18)

ROW 5 (19 : 29)

ROW 6 (29 : 33)

ROW 7 (34 : 40)

ROW 8 (41 : 49)

ROW 9 (49 : 56)

ROW 10 (56 : 63)

RRM

PROGRAM REGISTERS NEEDED: 15

ROW 1 (1 : 7)

ROW 2 (7 : 17)

ROW 3 (18 : 26)

ROW 4 (27 : 35)

ROW 5 (36 : 44)

ROW 6 (45 : 54)

ROW 7 (54 : 64)

ROW 8 (65 : 71)

PHN

PROGRAM REGISTERS NEEDED: 7

ROW 1 (1 : 6)

ROW 2 (7 : 16)

ROW 3 (16 : 24)

ROW 4 (25 : 29)

FAST

PROGRAM REGISTERS NEEDED: 11

ROW 1 (1 : 6)

ROW 2 (6 : 16)

ROW 3 (17 : 29)

ROW 4 (30 : 42)

ROW 5 (43 : 49)

ROW 6 (50 : 51)

LPAS

PROGRAM REGISTERS NEEDED: 93

ROW 1 (1 : 6)

ROW 2 (6 : 15)

ROW 3 (15 : 20)

ROW 4 (20 : 24)

ROW 5 (24 : 33)

ROW 6 (33 : 36)

ROW 7 (37 : 42)

ROW 8 (42 : 49)

ROW 9 (49 : 54)

ROW 10 (54 : 59)

ROW 11 (59 : 67)

ROW 12 (68 : 77)

ROW 13 (78 : 88)

ROW 14 (89 : 97)

ROW 15 (98 : 109)

ROW 16 (110 : 117)

ROW 17 (118 : 125)

ROW 18 (126 : 133)

ROW 19 (134 : 141)

ROW 20 (142 : 153)

ROW 21 (154 : 162)

ROW 22 (162 : 171)

ROW 23 (171 : 180)

ROW 24 (180 : 189)

ROW 25 (189 : 197)

ROW 26 (197 : 204)

ROW 27 (204 : 213)

ROW 28 (214 : 222)

ROW 29 (223 : 233)

ROW 30 (234 : 242)

ROW 31 (243 : 251)

ROW 32 (252 : 262)

ROW 33 (262 : 265)

ROW 34 (266 : 269)

ROW 35 (269 : 277)

ROW 36 (277 : 285)

ROW 37 (285 : 290)

ROW 38 (290 : 290)

ROW 39 (290 : 291)

ROW 40 (292 : 300)

ROW 41 (300 : 309)

ROW 42 (310 : 319)

ROW 43 (320 : 327)

ROW 44 (327 : 335)

ROW 45 (335 : 344)

ROW 46 (344 : 353)

ROW 47 (353 : 362)

ROW 48 (363 : 371)

ROW 49 (372 : 378)

ROW 50 (378 : 383)

CVPL

PROGRAM REGISTERS NEEDED: 121

ROW 1 (1 : 3)

ROW 2 (3 : 8)

ROW 3 (8 : 12)

ROW 4 (12 : 19)

ROW 5 (19 : 28)

ROW 6 (29 : 36)

ROW 7 (37 : 45)

ROW 8 (46 : 51)

ROW 9 (51 : 59)

ROW 10 (59 : 66)

ROW 11 (67 : 67)

ROW 12 (68 : 76)

ROW 13 (77 : 82)

ROW 14 (83 : 89)

ROW 15 (90 : 96)

ROW 16 (97 : 102)

ROW 17 (103 : 111)

ROW 18 (111 : 117)

ROW 19 (117 : 124)

ROW 20 (124 : 131)

ROW 21 (131 : 134)

ROW 22 (135 : 141)

ROW 23 (142 : 150)

ROW 24 (151 : 157)

ROW 25 (157 : 160)

ROW 26 (160 : 163)

ROW 27 (163 : 169)

ROW 28 (169 : 177)

ROW 29 (177 : 183)

ROW 30 (184 : 190)

ROW 31 (190 : 193)

ROW 32 (193 : 195)

ROW 33 (195 : 199)

ROW 34 (199 : 202)

ROW 35 (203 : 207)

ROW 36 (208 : 215)

ROW 37 (216 : 223)

ROW 38 (223 : 231)

ROW 39 (232 : 241)
ROW 40 (242 : 246)
ROW 41 (247 : 255)
ROW 42 (256 : 264)
ROW 43 (265 : 273)
ROW 44 (274 : 278)
ROW 45 (279 : 282)
ROW 46 (282 : 287)
ROW 47 (288 : 296)
ROW 48 (296 : 304)
ROW 49 (305 : 311)
ROW 50 (312 : 323)
ROW 51 (323 : 330)
ROW 52 (331 : 337)
ROW 53 (338 : 345)
ROW 54 (345 : 350)
ROW 55 (351 : 358)
ROW 56 (358 : 367)
ROW 57 (367 : 375)
ROW 58 (375 : 383)
ROW 59 (384 : 394)
ROW 60 (395 : 402)
ROW 61 (403 : 410)
ROW 62 (410 : 417)
ROW 63 (418 : 426)
ROW 64 (426 : 433)
ROW 65 (434 : 438)

SC

PROGRAM REGISTERS NEEDED: 74

ROW 1 (1 : 4)
ROW 2 (4 : 9)
ROW 3 (10 : 17)
ROW 4 (18 : 23)
ROW 5 (24 : 30)
ROW 6 (31 : 37)
ROW 7 (37 : 43)
ROW 8 (43 : 49)
ROW 9 (50 : 57)
ROW 10 (58 : 63)
ROW 11 (64 : 70)
ROW 12 (70 : 76)
ROW 13 (77 : 82)
ROW 14 (82 : 85)
ROW 15 (85 : 89)
ROW 16 (90 : 94)
ROW 17 (94 : 97)
ROW 18 (97 : 103)
ROW 19 (103 : 106)
ROW 20 (106 : 111)
ROW 21 (111 : 116)
ROW 22 (117 : 121)
ROW 23 (121 : 124)
ROW 24 (125 : 130)
ROW 25 (130 : 133)
ROW 26 (134 : 139)
ROW 27 (139 : 143)
ROW 28 (144 : 148)
ROW 29 (148 : 151)
ROW 30 (152 : 157)
ROW 31 (157 : 161)
ROW 32 (162 : 166)
ROW 33 (166 : 169)
ROW 34 (169 : 172)
ROW 35 (173 : 178)
ROW 36 (178 : 181)
ROW 37 (181 : 187)
ROW 38 (187 : 190)
ROW 39 (190 : 196)
ROW 40 (196 : 198)

SC DEMO

PROGRAM REGISTERS NEEDED: 45

ROW 1 (1 : 2)

ROW 2 (2 : 3)
ROW 3 (3 : 5)
ROW 4 (5 : 6)
ROW 5 (6 : 7)
ROW 6 (7 : 11)
ROW 7 (11 : 15)
ROW 8 (15 : 18)
ROW 9 (18 : 22)
ROW 10 (22 : 25)
ROW 11 (26 : 31)
ROW 12 (31 : 39)
ROW 13 (40 : 48)
ROW 14 (49 : 55)
ROW 15 (55 : 61)
ROW 16 (61 : 69)
ROW 17 (69 : 75)
ROW 18 (76 : 81)
ROW 19 (81 : 86)
ROW 20 (86 : 94)
ROW 21 (94 : 100)
ROW 22 (101 : 109)
ROW 23 (110 : 116)
ROW 24 (116 : 122)

COMP

PROGRAM REGISTERS NEEDED: 17

ROW 1 (1 : 8)
ROW 2 (9 : 21)
ROW 3 (22 : 34)
ROW 4 (35 : 47)
ROW 5 (48 : 60)
ROW 6 (61 : 73)
ROW 7 (74 : 86)
ROW 8 (87 : 99)
ROW 9 (100 : 107)

SUB1

PROGRAM REGISTERS NEEDED: 13

ROW 1 (1 : 5)

ROW 2 (5 : 13)
ROW 3 (14 : 23)
ROW 4 (24 : 34)
ROW 5 (34 : 43)
ROW 6 (43 : 54)
ROW 7 (55 : 61)

LBW

PROGRAM REGISTERS NEEDED: 21

ROW 1 (1 : 5)
ROW 2 (5 : 8)
ROW 3 (9 : 15)
ROW 4 (16 : 24)
ROW 5 (25 : 34)
ROW 6 (35 : 43)
ROW 7 (44 : 50)
ROW 8 (51 : 59)
ROW 9 (59 : 61)
ROW 10 (61 : 67)
ROW 11 (67 : 71)
ROW 12 (71 : 71)

COMP

PROGRAM REGISTERS NEEDED: 8

ROW 1 (1 : 2)
ROW 2 (2 : 11)
ROW 3 (11 : 21)
ROW 4 (22 : 30)
ROW 5 (31 : 31)

VK

PROGRAM REGISTERS NEEDED: 36

ROW 1 (1 : 5)
ROW 2 (6 : 8)
ROW 3 (8 : 10)
ROW 4 (11 : 17)
ROW 5 (17 : 22)
ROW 6 (23 : 31)
ROW 7 (32 : 40)
ROW 8 (41 : 46)

ROW 9 (46 : 53)
ROW 10 (54 : 56)
ROW 11 (56 : 62)
ROW 12 (62 : 69)
ROW 13 (70 : 79)
ROW 14 (80 : 89)
ROW 15 (90 : 95)
ROW 16 (95 : 101)
ROW 17 (101 : 108)
ROW 18 (109 : 116)
ROW 19 (117 : 122)

AORD;ACMP

PROGRAM REGISTERS NEEDED: 21

ROW 1 (1 : 5)
ROW 2 (6 : 13)
ROW 3 (14 : 20)
ROW 4 (20 : 27)
ROW 5 (27 : 31)
ROW 6 (31 : 34)
ROW 7 (35 : 43)
ROW 8 (43 : 52)
ROW 9 (52 : 59)
ROW 10 (59 : 62)
ROW 11 (63 : 68)

FCT

PROGRAM REGISTERS NEEDED: 14

ROW 1 (1 : 2)
ROW 2 (3 : 9)
ROW 3 (9 : 11)
ROW 4 (11 : 16)
ROW 5 (16 : 21)
ROW 6 (21 : 26)
ROW 7 (26 : 29)
ROW 8 (30 : 30)

SUB2

PROGRAM REGISTERS NEEDED: 8

ROW 1 (1 : 4)
ROW 2 (4 : 10)
ROW 3 (11 : 15)
ROW 4 (15 : 22)
ROW 5 (23 : 23)

ISX

PROGRAM REGISTERS NEEDED: 23

ROW 1 (1 : 5)
ROW 2 (5 : 10)
ROW 3 (11 : 15)
ROW 4 (15 : 19)
ROW 5 (20 : 29)
ROW 6 (30 : 37)
ROW 7 (38 : 45)
ROW 8 (46 : 53)
ROW 9 (54 : 64)
ROW 10 (65 : 71)
ROW 11 (72 : 81)
ROW 12 (81 : 89)
ROW 13 (89 : 89)

IRX

PROGRAM REGISTERS NEEDED: 32

ROW 1 (1 : 5)
ROW 2 (5 : 14)
ROW 3 (15 : 19)
ROW 4 (20 : 23)
ROW 5 (24 : 28)
ROW 6 (29 : 37)
ROW 7 (38 : 46)
ROW 8 (46 : 53)
ROW 9 (54 : 62)
ROW 10 (62 : 70)
ROW 11 (71 : 78)
ROW 12 (78 : 82)
ROW 13 (82 : 87)
ROW 14 (88 : 97)
ROW 15 (97 : 105)
ROW 16 (106 : 110)

ROW 17 (110 : 112)

ALFA TN

PROGRAM REGISTERS NEEDED: 55

ROW 1 (1 : 2)
ROW 2 (3 : 9)
ROW 3 (10 : 17)
ROW 4 (17 : 24)
ROW 5 (24 : 31)
ROW 6 (32 : 38)
ROW 7 (39 : 45)
ROW 8 (46 : 52)
ROW 9 (52 : 57)
ROW 10 (57 : 62)
ROW 11 (63 : 68)
ROW 12 (68 : 73)
ROW 13 (74 : 79)
ROW 14 (79 : 84)
ROW 15 (85 : 90)
ROW 16 (90 : 95)
ROW 17 (96 : 101)
ROW 18 (102 : 107)
ROW 19 (107 : 112)
ROW 20 (112 : 117)
ROW 21 (117 : 122)
ROW 22 (122 : 127)
ROW 23 (128 : 133)
ROW 24 (133 : 137)
ROW 25 (137 : 141)
ROW 26 (142 : 146)
ROW 27 (146 : 150)
ROW 28 (150 : 154)
ROW 29 (155 : 159)
ROW 30 (160 : 160)

DIAL

PROGRAM REGISTERS NEEDED: 32

ROW 1 (1 : 4)
ROW 2 (4 : 7)
ROW 3 (7 : 11)
ROW 4 (11 : 14)
ROW 5 (14 : 18)
ROW 6 (18 : 21)
ROW 7 (21 : 24)
ROW 8 (24 : 25)
ROW 9 (25 : 28)
ROW 10 (29 : 30)
ROW 11 (30 : 33)
ROW 12 (33 : 39)
ROW 13 (40 : 48)
ROW 14 (49 : 56)
ROW 15 (57 : 64)
ROW 16 (64 : 72)
ROW 17 (72 : 78)

KA

PROGRAM REGISTERS NEEDED: 96

ROW 1 (1 : 5)
ROW 2 (6 : 12)
ROW 3 (12 : 17)
ROW 4 (18 : 26)
ROW 5 (26 : 32)
ROW 6 (33 : 40)
ROW 7 (41 : 48)
ROW 8 (48 : 57)
ROW 9 (58 : 68)
ROW 10 (69 : 77)
ROW 11 (78 : 84)
ROW 12 (84 : 89)
ROW 13 (89 : 95)
ROW 14 (96 : 103)
ROW 15 (103 : 114)
ROW 16 (115 : 124)
ROW 17 (124 : 129)
ROW 18 (129 : 134)

ROW 19 (134 : 141)

ROW 20 (141 : 144)

ROW 21 (145 : 151)

ROW 22 (151 : 156)

ROW 23 (157 : 165)

ROW 24 (165 : 171)

ROW 25 (172 : 177)

ROW 26 (178 : 184)

ROW 27 (185 : 191)

ROW 28 (192 : 196)

ROW 29 (196 : 202)

ROW 30 (203 : 209)

ROW 31 (209 : 216)

ROW 32 (217 : 225)

ROW 33 (225 : 228)

ROW 34 (228 : 231)

ROW 35 (232 : 240)

ROW 36 (241 : 247)

ROW 37 (247 : 249)

ROW 38 (250 : 258)

ROW 39 (259 : 267)

ROW 40 (267 : 273)

ROW 41 (273 : 281)

ROW 42 (281 : 287)

ROW 43 (288 : 296)

ROW 44 (296 : 304)

ROW 45 (304 : 311)

ROW 46 (312 : 316)

ROW 47 (316 : 323)

ROW 48 (323 : 329)

ROW 49 (329 : 337)

ROW 50 (338 : 345)

ROW 51 (346 : 349)

ROW 52 (349 : 351)

MKA

PROGRAM REGISTERS NEEDED: 134

ROW 1 (1 : 5)

ROW 2 (5 : 8)

ROW 3 (8 : 14)

ROW 4 (15 : 18)

ROW 5 (19 : 25)

ROW 6 (26 : 30)

ROW 7 (31 : 38)

ROW 8 (39 : 42)

ROW 9 (42 : 47)

ROW 10 (47 : 54)

ROW 11 (54 : 58)

ROW 12 (59 : 69)

ROW 13 (70 : 79)

ROW 14 (80 : 87)

ROW 15 (88 : 97)

ROW 16 (98 : 106)

ROW 17 (106 : 111)

ROW 18 (111 : 118)

ROW 19 (118 : 122)

ROW 20 (123 : 127)

ROW 21 (127 : 132)

ROW 22 (132 : 137)

ROW 23 (138 : 143)

ROW 24 (143 : 148)

ROW 25 (148 : 154)

ROW 26 (155 : 156)

ROW 27 (156 : 161)

ROW 28 (161 : 169)

ROW 29 (169 : 173)

ROW 30 (173 : 176)

ROW 31 (176 : 182)

ROW 32 (183 : 188)

ROW 33 (188 : 194)

ROW 34 (195 : 201)

ROW 35 (201 : 207)

ROW 36 (207 : 214)

ROW 37 (215 : 221)

ROW 38 (221 : 226)

ROW 39 (227 : 234)

ROW 40 (235 : 241)

ROW 41 (241 : 247)

ROW 42 (247 : 251)

ROW 43 (251 : 260)

ROW 44 (261 : 266)

ROW 45 (266 : 272)

ROW 46 (272 : 278)

ROW 47 (279 : 285)

ROW 48 (285 : 289)

ROW 49 (290 : 295)

ROW 50 (295 : 301)

ROW 51 (302 : 306)

ROW 52 (306 : 310)

ROW 53 (311 : 317)

ROW 54 (317 : 325)

ROW 55 (326 : 329)

ROW 56 (330 : 336)

ROW 57 (337 : 340)

ROW 58 (341 : 347)

ROW 59 (348 : 354)

ROW 60 (354 : 360)

ROW 61 (360 : 366)

ROW 62 (367 : 372)

ROW 63 (373 : 378)

ROW 64 (378 : 385)

ROW 65 (385 : 392)

ROW 66 (393 : 399)

ROW 67 (400 : 406)

ROW 68 (406 : 409)

ROW 69 (409 : 417)

ROW 70 (418 : 420)

ROW 71 (421 : 428)

ROW 72 (429 : 437)

ROW 73 (437 : 437)

INDEX

This index should be used in conjunction with the Glossary - Appendix G (pages 187, 227, 233, 251, and 273) and the routines themselves. The latter may be referenced by using the Pocket Guide. Index space is not taken up with the 122 routine titles. The order of this index is alphabetical with numbers following the letters and not in the HEX table order that is used throughout this manual. Page numbers preceded by letters refer to appendices.

A

B

C

D

T

U

V

W

X

Record here any entries that you are unable to locate in the index. These will be collected for a future revision to the index.

NOTES

END **END** **END** **END** **END**

PPC ROM USER'S MANUAL

[×] SYNTHETIC PROGRAMMING [÷] ALPHA X □
RPN □ BARCODE [A] R-P CONVERSION □ LCD
□ HOUSEKEEPING ROUTINES □ PRINTERS [×]
DATA REGISTERS □ USER DEFINABLE KEY
□ MODULE [π] SERIAL BUS □ ANNUNCIATORS □
XROM □ KEYSTROKE □ OVERLAY □ MATRIX
□ PERIPHERAL □ CONTINUOUS MEMORY □
BLOCK CLEAR [π] PLOTTING [×] CARD READER
□ GLOBAL LABEL □ INDIRECT ADDRESS □
SCRATCH REGISTERS □ SOFTWARE □ STO M
[÷] ENTER IS GREATER THAN EQUALS [A] X>Y □
PPC □ SWITCH WITHOUT REPLACEMENT □ π
□ 82506A [÷] POLYNOMIAL SOLUTION □ IO □
RANDOM NUMBER □ N-I-PV-PMT-FV □ 41
□ QUAD □ MATHEMATICS □ BYTE JUMPER [÷]
SUBROUTINES □ STACK □ FS?C □ 82500A07
□ X-Y-Z-T [×] STATUS CARD □ TONES 1-127 □
MICROCODING [÷] ŁUKASIEWICZ [×] ASSIGN □ ⊢
[π] CURTAIN RAISER [+] ALPHA STRING [C] M4 □
DEBUGGING □ XEQ □ RJN [−] RECALL □ USER
□ REVERSE POLISH NOTATION □ CATALOG □
PERSONAL PROGRAMMING □ SHIFT □ WAND
[+] BACK ARROW □ APPEND □ GRAPICS □ ÷ [π]
HEWLETT-PACKARD □ "**" [−] STATISTICAL
[A] CONDITIONAL TESTS □ ROM □ ENTER↑ □
SANTA ANA [÷] 319 □ EXPANSION □ PORTS
□ PROMPTING □ LOCAL LABEL □ ARRAYS □
EPROM BOX □ BRANCHING [A] SST □ TRACE
□ STORAGE MEDIA [π] SDS □ CHAPTERS [×]
EDITING □ PPC LOGO □ MASK [D] 8K [×] 5000
[÷] EMULATOR □ GEOMETRY □ SOLUTIONS □
A STANDARD FOR THE PERSONAL USER OF
□ THE HP-41C, HP-41CV & PERIPHERALS □

REVERSE POLISH NOTATION

[+] [−] [×] [÷] [+] [−] [×] [÷] [+] [−] [×] [÷] [+] [−] [×] [÷] [+] [−] [×] [÷] [+]