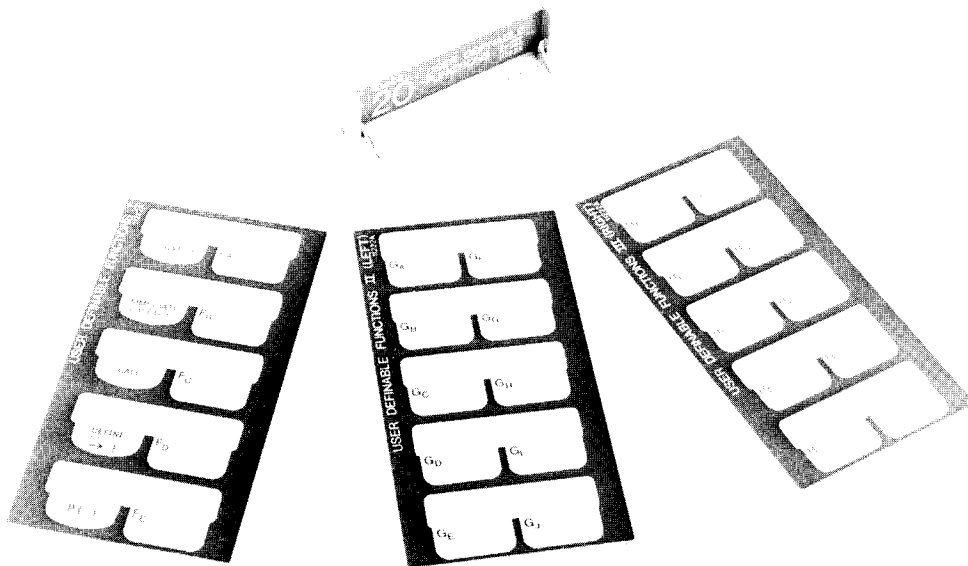




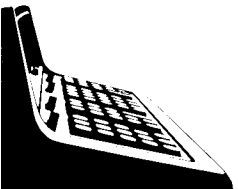
 **HEWLETT-PACKARD 9820A CALCULATOR
11222A USER DEFINABLE FUNCTIONS BLOCK**



11222A USER DEFINABLE FUNCTIONS BLOCK

HEWLETT-PACKARD CALCULATOR PRODUCTS DIVISION

P.O. Box 301, Loveland, Colorado 80537, Tel. (303) 667-5000
Rue du Bois-du-Lan 7, CH-1217 Meyrin 2, Geneva, Tel. (022) 41 54 00





WHO NEEDS THE BLOCK?

The User Definable Functions Block brings a great deal of extra power to the 9820A Calculator, although, on first acquaintance with the block, it perhaps is not at all obvious what that power might be, or why one would even need it.

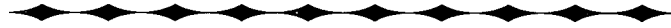
The User Definable Functions Block enables up to 25 independent subprograms[†] to be stored in the calculator at one time. Subprograms can be of three types:

- Subroutines, which enable the programmer to more easily segment his programs and so make more efficient use of his time.
- Functions, which can extend the language to include any special mathematical functions in some specific discipline, functions perhaps not normally available on a calculator.
- Procedures, which can be used to dedicate the calculator to perform specific often-required tasks.

Following is a brief description of the special features in this block that enable the three types of subprogram to be such powerful tools.

- Each subprogram is named by one key; to refer to a particular subprogram, just press its key — this saves having to spell out literals, or 'go to' line numbers. If the first statement in the subprogram is itself a literal, then that literal (without its quotation marks) actually appears in the display when the subprogram's key is pressed.
- Each subprogram is entirely independent from each other subprogram — you do not need to remember which subprograms use which line numbers; nor do you have to change other subprograms whenever you lengthen or shorten one of them.
- 'Parameter Passing' — this is an extremely powerful feature, normally available only to the computer programmer. This feature enables you to write subprograms using 'dummy variables' instead of actual values and register names. When you use the subprogram you specify its 'parameters' — which values and register names are to correspond to the dummy variables in that subprogram — for that time only. Even though you may well specify a different set of parameters each time you use that subprogram, you need not actually change the subprogram.

[†] The word 'subprogram' is used to denote any program defined by the User Definable Functions Block.



Parameter passing eliminates the need for most of the time-consuming and error-prone bookkeeping normally associated with program writing. You no longer need to ask such questions as, 'Has that register been reserved for some other subprogram? Will this higher-numbered register become part of the program area? If I use this register, will it split that block of consecutive registers that I am saving for my statistics subprograms?'

Parameter passing also saves space, in both the program area and data registers, by eliminating some of the cumbersome programming and register manipulating previously required before branching to subroutines. It is no longer necessary to move data to special registers before branching, or to take the results from other registers after returning to the mainline program. Now, one simple 'call' statement, with a parameter list, controls the branch and tells the subprogram where to 'look' for its data and where to put the results afterwards.

- The mathematical functions are written just like any other program, yet they are used in a totally different way — a function and its argument can be inserted directly into an expression, in exactly the same way, for instance, as the square root function and its argument is inserted directly into an expression. Defined functions can also have multi-parameter arguments.

Functions enable you to orient your calculator towards some specific discipline, by storing whichever functions you select — N-factorial, or the square root of the sum of two (or more) squares for example; or, if you already have the Mathematics ROM, you could extend its scope by adding the hyperbolic and inverse-hyperbolic functions. In fact, if your work is that specialized, you could even have a 'Gudermannian' key [$\text{gd } x = \tan^{-1}(\sinh x)$].

- The 'immediate execute' feature is associated with the procedure subprograms. This feature enables the calculator to be dedicated to perform specific often-required tasks, such as calculating mortgage payments, or calculating the stresses in a bridge truss. The power of the immediate execute feature lies in that it enables the subprograms to be so easily initialized and started — simply by pressing one key. As well as saving time, this simplifies operation of the calculator to the point where even highly complex calculations can be run by personnel who have little or no special training, either in your discipline or in programming.

- Complete flexibility is retained, even though your calculator is dedicated to specific tasks. Subprograms can be recorded on magnetic cards, and reloaded into the calculator, just as easily as can any other program. The calculator can be rapidly 'switched' from one discipline to another whenever the need arises.

TABLE OF CONTENTS

PREFACE	ii
CHAPTER 1. GENERAL INFORMATION	
DESCRIPTION	1-1
Supplied Equipment	1-1
INSPECTION PROCEDURE	1-1
UDF BLOCK INSTALLATION	1-1
The Key Overlays	1-1
Selecting the Slot	1-2
Installing the Block	1-2
Memory Conversion	1-2
CHAPTER 2. CONCEPTS OF THE BLOCK	
THREE TYPES OF SUBPROGRAMS	2-1
Subroutines	2-1
Functions	2-1
Procedures	2-1
PARAMETERS AND VARIABLES	2-1
Dummy Variables	2-2
Local Variables	2-2
PARAMETER PASSING	2-2
P-Number Independence	2-3
USE OF THE MEMORY	2-3
The Memory Map	2-3
The Definable Area	2-4
CHAPTER 3. WRITING SUBPROGRAMS	
THE KEY MNEMONICS	3-1
WHICH TYPE?	3-1
GENERAL CONSIDERATIONS	3-1
THE PROCEDURE SUBPROGRAM	3-2
THE SUBROUTINE SUBPROGRAM	3-3
THE FUNCTION SUBPROGRAM	3-3
NAMING SUBPROGRAMS	3-4
NESTING 'ENTER' STATEMENTS – A DON'T	3-5

TABLE OF CONTENTS



CHAPTER 3. WRITING SUBPROGRAMS (CONTINUED)

SYNTAXES – A SUMMARY	3-5
P-Numbers	3-5
The Value of a Function	3-6
The 'IMMEDIATE EXECUTE' Statement	3-6
The 'END' Statement	3-6
The 'RETURN' Statement	3-6

CHAPTER 4. MECHANICS OF OPERATION

ESTABLISHING A HEADING	4-1
The Subroutine/Procedure	4-1
The Function	4-1
SUBPROGRAM STORAGE	4-1
Accessing a Subprogram	4-1
Storing A Subprogram	4-1
Listing a Subprogram	4-2
EDITING SUBPROGRAMS	4-2
Recalling and Modifying	4-2
The 'SCRATCH' Statement	4-2
RUNNING SUBPROGRAMS	4-2
Using Procedure Subprograms	4-2
The 'CALL' Statement	4-3
Using Subroutine Subprograms	4-3
Using Function Subprograms	4-3
RECORDING SUBPROGRAMS	4-4
General Considerations	4-4
Single Subprograms	4-5
A Series of Subprograms	4-5
Loading Under Program-Control	4-5
DIAGNOSTIC NOTES	4-6

CHAPTER 5. ADDITIONAL INFORMATION

THE 'IMMEDIATE EXECUTE' MODE	5-1
'CALLING' LOCAL SUBROUTINES	5-1
CAUSES OF NOTE 09	5-1

TABLE OF CONTENTS



APPENDIX

KEY MNEMONICS

KEY OVERLAYS

FIGURES

2-1. The Memory Map	2-3
2-2. The Definable Area	2-4

TABLE

1-1. Equipment Supplied	1-1
-----------------------------------	-----

Chapter 1

GENERAL INFORMATION

DESCRIPTION

The Hewlett-Packard 11222A User Definable Functions Block (the UDF Block) consists of a Read-Only Memory (ROM) and three key overlays. The ROM, which is for use in an HP 9820A Calculator, enables the user to assign his own unique meaning to each of up to 25 keys, by storing (up to) 25 independent subprograms at any one time. The ROM plugs into any of the three slots on top of the calculator; the three key overlays are then used to indicate the meanings given to the keys by the block. Space is left on those keys that the user can define so that he can, if he wishes, write his own definition on them. A more complete description of the block is contained in the preface to this manual. The three overlays, and the mnemonic associated with each key, are shown on the foldout at the back of this manual.

SUPPLIED EQUIPMENT

The items supplied with each UDF Block are listed in Table 1-1.

Table 1-1. Equipment Supplied

QTY.	DESCRIPTION	-hp- PART NO.
1	Key Overlay I	7120-1689
1	Key Overlay II (LEFT)	7120-1690
1	Key Overlay III (RIGHT)	7120-1691
2	Operating Manual	09820-90022

INSPECTION PROCEDURE

The UDF Block was carefully inspected, both mechanically and electrically, before it was shipped to you. Inspect the block for physical damage and also check that the equipment listed in Table 1-1 is present.

To check operation of the block, refer to the Model 20 System Electrical Inspection Booklet,

supplied with each calculator. Installation procedures are given below.

If there is any damage or electrical malfunction, contact your nearest HP Sales and Service office — addresses are provided at the back of this manual.

UDF BLOCK INSTALLATION

THE KEY OVERLAYS

Like most ROM blocks, the UDF Block can be installed in any of the three numbered slots on top of the calculator. As well as defining the keyblock immediately in front of the slot in which it is installed, the UDF Block also defines the keyblock in front of any empty slot — hence the requirement for three different overlays: I, II (LEFT), and III (RIGHT), shown on the foldout at the back of this manual.

NOTE

The terms 'left' and 'right' correspond, respectively, to the user's left and right when facing the keyboard.

- a. Regardless of which slot contains the UDF Block, the I-overlay is always installed over the half-keys immediately in front of that slot.

UDF BLOCK INSTALLATION

THE KEY OVERLAYS (cont'd)

- If there is only one other ROM block present, then the II-overlay is installed over the half-keys in front of the empty slot. (The II-overlay, then, might be to the left or to the right of the I-overlay, and might or might not be adjacent to it.)
- If there is no other ROM block present when the UDF Block is installed, then, of the two empty slots, the II-overlay is installed over the half-keys in front of the left-most empty slot, and the III-overlay over the half-keys in front of the right-most empty slot.

SELECTING THE SLOT

Even though a block can be installed in any of the three slots, programs recorded on magnetic cards dictate that any required block be in a specific slot — namely, the same slot that the block was in when the program was recorded. Before loading any program from a magnetic card, always check that program's user instructions to determine which ROM's should be installed in which slots (and, in the case of the UDF Block, if any slots must remain empty) for that program.

As far as is practical, programs published by HP require the UDF ROM to be in Slot #1.

INSTALLING THE BLOCK

To install the block:

- Switch the calculator off — if you do inadvertently leave the calculator on, an installed block will not be 'accepted' until MEMORY ERASE is pressed.
- Position the block vertically (with its label 'right-side-up' when viewed from the front of the calculator) over the trap-door of the selected slot. Push the block through the trap-door, straight down into the slot, until it is firmly seated.
- (Refer to 'The Key Overlays', above, to determine which of the three overlays to install and where to install them.) To install an overlay, insert the tab at the top of the overlay into the locking-slot at the head of the selected key-block; then press the overlay down over the keys.
- The block is now installed; switch the calculator on and the null program will appear in the display:

```
0: END F
```

MEMORY CONVERSION

When the UDF Block is installed, it automatically converts 83 words of User Read-Write-Memory to Internal Read-Write-Memory. This is equivalent to the loss of 21 (actually 20¾) R registers, which will be reflected in the R number appearing at the end of any program listing. (The requirement for 21 registers does not change even though more than one keyblock is defined by the UDF ROM.)

Chapter 2

CONCEPTS OF THE BLOCK

◆◆◆◆◆ THREE TYPES OF SUBPROGRAMS ◆◆◆◆◆

Even though the name of the block uses the words 'definable functions', there is far more available with the UDF Block than just 'functions' in the mathematical sense of the word. The description of the block, in the preface to this manual, indicates that the definable keys can be made to represent any of three types of subprogram — subroutine subprograms, function subprograms, and procedure subprograms. (In this manual the word 'subprogram' denotes any program directly associated with the UDF Block, unless indicated otherwise.)

SUBROUTINES

Subroutine subprograms are, in general, subject to the same syntax considerations as are local subroutines described in the calculator's operating and programming manual. However, the subroutine subprogram can be written using dummy and local variables (described below), instead of using real variables. The variables are then made to correspond to actual values and registers, each time the subprogram is called, by means of a parameter list (the list can be different, if required, for each call).

FUNCTIONS

Function subprograms are the 'definable functions' described briefly, under the heading 'functions', in Chapter 4 of the calculator's operating and programming manual. These are functions in

the mathematical sense of the word, and they are used as such.

Functions are subject to essentially the same syntax as is any supplied function (the square root function, for example). However, a function subprogram can have, if necessary, an argument containing more than one parameter; multiple-parameter arguments are not possible with the function on the basic calculator (the square root function) or with the functions supplied by the Math ROM (HP 11221A Mathematics ROM). Like the subroutine subprogram, function subprograms can be written using dummy and local variables; the dummy variable(s) then corresponds to the argument of the function.

PROCEDURES

Procedure subprograms are the same as any main-line program except that the procedure can be executed 'immediately'. That is, as soon as the key representing a procedure subprogram is pressed, that subprogram is immediately started, without any other key having to be pressed. Unlike subroutine subprograms and function subprograms, procedures cannot have dummy or local variables. They must have real variables (actual register names) and known constants. (Dummy variables cannot be used because it is not possible to write a list of parameters after pressing a key that causes 'immediate execution'.)

◆◆◆◆◆ PARAMETERS AND VARIABLES ◆◆◆◆◆

One of the most important features of the UDF Block is that of 'parameter passing'. This enables subroutine subprograms and function subprograms to be written using 'unknown' variables. Values or register names are then assigned to the variables (by means of a parameter list) each time the subprogram is used.

There are two types of 'unknown' variable: 'dummy' variables and 'local' variables. For our

purposes, the only absolute distinction between a dummy variable and a local variable is that a dummy variable is assigned a value by the user, when the subprogram is used, whereas a local variable is not. As will be seen, there are some other 'obvious' differences but they are not necessarily hard and fast. The variables are designated by P-numbers (P(value)) — P1, P2, etc. (see 'Parameter Passing', below, for further explanation).

PARAMETERS AND VARIABLES

DUMMY VARIABLES

In general, dummy variables represent the input and output data for the subprogram; whenever the subprogram is used, a list of parameters assigns the values and register names that are to correspond to the dummy variables. For example, when calling a subroutine subprogram designed to calculate N-factorial (N!), the parameter list would most probably contain two parameters: an input datum and an output datum. In this example, the input datum is the value for N (the number whose factorial is to be calculated); its parameter could be anything that has a definite value — a constant, an expression, a register name, etc. Of necessity, the parameter for the output datum can be only a register name; that is, the result has to be stored somewhere before the return is made to the calling program (assuming that the result is not simply to be printed and then forgotten). 'CALL

N! 6,A', for example, would cause a branch to the N! subprogram; the factorial of 6 would then be calculated and the result stored in register A.

LOCAL VARIABLES

Local variables can be considered as the intermediate results of the subprogram and, as such, do not require parameters. They represent working registers (or temporary storage) for those values that are generated during the subprogram, that are required for (at least part of) the remainder of the subprogram, and that are of no further interest once execution of the subprogram is completed. For example, in the N! subprogram described above, the multiplier (1, then 2, then 3, etc.) used to generate each partial result is purely a local variable and is of no consequence once the final value is calculated.

PARAMETER PASSING

The act of assigning values to dummy variables, by means of a parameter list, is known as 'parameter passing'.

The variables in a subprogram (both dummy variables and local variables) are designated by means of P-numbers: the 'P(' key followed by a number — P1, P2, P3, etc. (P0 does not exist). When the subprogram is used, the parameters in the list correspond, in turn, to the P-numbers in numerical order — the first parameter in the list to P1, the second to P2, and so on, regardless of the order in which the P-numbers are used in the subprogram. For example, instead of writing $\sqrt{(AA + BB)} \rightarrow C$, you could write $\sqrt{(P2P2 + P1P1)} \rightarrow P3$ (the transposition of P1 and P2 is deliberate). If this is (the essential part of) your subprogram, then, obviously, P1, P2 and P3 are all dummy variables and must have corresponding parameters.

A typical parameter list for this subprogram might be 'A, B, C' (commas must be included to separate the parameters). This list assigns registers A, B and C to be used, respectively, in place of P1, P2 and P3 — notice that A, being the first parameter, is assigned to P1 even though P2 appears first in the subprogram. The next time the subprogram is used, perhaps the same parameter list will be

passed, or perhaps it will be changed completely (say to '6, 3X/4Y, R16').

To illustrate the concept of a subprogram with a local variable, the above example subprogram could be expanded to $\sqrt{(P2P2 + P1P1)} \rightarrow P4$; $P4/2 \rightarrow P3$. In this case P4 is an intermediate result (a local variable) and need not be included in the parameter list. However, if for some reason you did wish to pass a parameter to P4 then you could do so, by adding a fourth parameter to the list — by definition, P4 would then become a dummy variable.

Local variables, which do not have parameters passed to them, must all have higher P-numbers than must the dummy variables, which do have associated parameters. The P-numbers used to designate variables should be consecutive and should start with P1. If any numbers are missing in the P-numbers used to designate dummy variables, then the parameters corresponding to the missing numbers will simply be ignored. If any numbers are missing in the local variables, then more working registers than necessary will be established for that call; this may be undesirable (see 'The Memory Map', in this chapter).

PARAMETER PASSING

P-NUMBER INDEPENDENCE

Stored subprograms are entirely independent of each other (unless they are deliberately linked by means of certain programming techniques). Therefore, a P-number used in one subprogram bears no relationship to the same P-number used in any other subprogram. However, it can be made to do

so: for example, suppose P2 appears in two subprograms and the parameter passed to P2 is, in both cases, R4; or suppose that one subprogram calls another and the third parameter in the list happens to be P3. (One subprogram calling another is known as 'nesting' subprograms and is subject to essentially the same rules as is nesting local subroutines in a mainline program.)

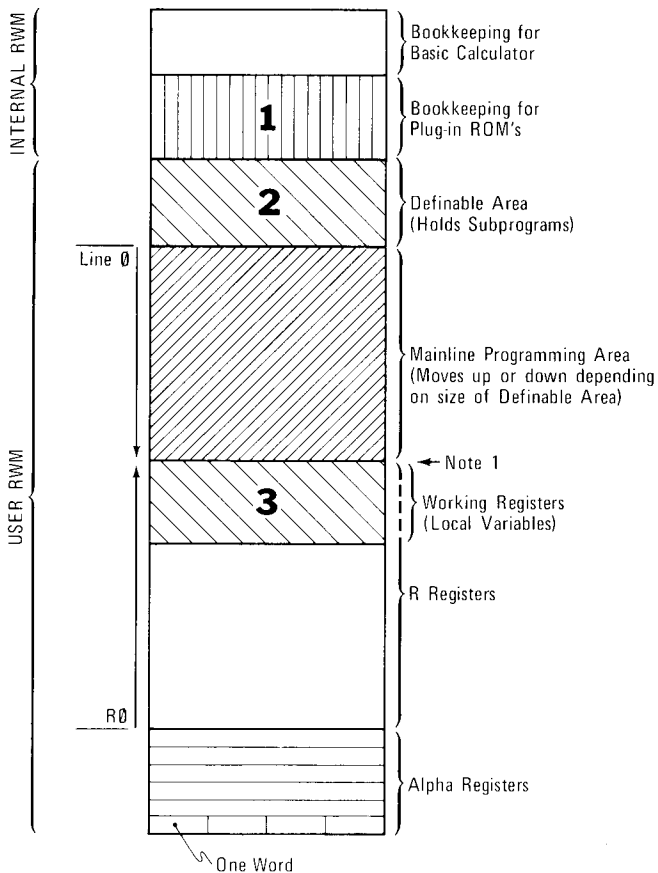
USE OF THE MEMORY

THE MEMORY MAP

Figure 2-1 is a map of the calculator's memory, showing how it is used when the UDF Block in User's RWM automatically converted to Internal RWM when the block is installed (see Chapter 1).

The area marked '1 - Bookkeeping for Plug-in ROM's' contains the 83 words (20¾ registers) of User's RWM automatically converted to Internal RWM when the block is installed (see Chapter 1).

The area marked '2 - Definable Area' holds stored subprograms. This area, which is of zero size until a subprogram is stored, automatically increases and decreases in size, as subprograms are stored and 'scratched' (deleted), so as to (exactly) accommodate the subprograms currently stored. The mainline program area, together with whatever programs it contains, automatically moves up and down to reflect changes in the size of the definable area; the number of available R registers also changes accordingly. The definable area is entirely separate from the mainline area; subprograms cannot be accidentally accessed, deleted, or changed while mainline programs are being stored or edited.



NOTE

MEMORY ERASE (or turn-on) will erase all subprograms stored in the definable area; the area is part of User's RWM and, as such, is volatile.

NOTE 1
This boundary moves according to the amount of programming.

Figure 2-1. The Memory Map

The third area shown in Figure 2-1 ('3 - Working Registers') is used for the local variables in the subprograms. For most practical purposes, this area can be considered as existing only while a subprogram is running, and then not at all if the subprogram has no local variables. The area will be only as large as necessary to accommodate all local variables for a currently-running subprogram. The area used for local variables is also available to the R registers; the user must remember this

USE OF THE MEMORY

THE MEMORY MAP (cont'd)

when storing data into the higher-numbered R registers. Because programming usually ends somewhere in the middle of an R register, and the working registers start where the programming ends, the working registers and the R registers will rarely coincide; therefore, if, after running a subprogram with local variables, you display the contents of the highest available R registers, you will probably see meaningless numbers.

The area used by the dummy variables depends upon the type of parameter passed. When the parameter is a register name, then that actual register is used during execution of the subprogram; if the subprogram changes the value in that register (e.g., $2P2 \rightarrow P2$) then that register will retain its changed value after completion of the subprogram. When the parameter passed is a value that is not a register name (i.e., is a number or an expression), then the 'bookkeeping' area (not otherwise available to the user) is used; once the subprogram is completed, such values are lost.

THE DEFINABLE AREA

Each stored subprogram is represented by one of the definable keys, F_A through F_E (also G_A through G_J and H_A through H_J , if available – see 'UDF Block Installation', in Chapter 1). A key cannot represent more than one subprogram at a time.

Up to 25 subprograms can be stored in the definable memory at the same time, but they do not necessarily have to be in any specific order. As an example, Figure 2-2 shows the definable area when subprograms F_D , F_B and F_C have just been stored, in that order.

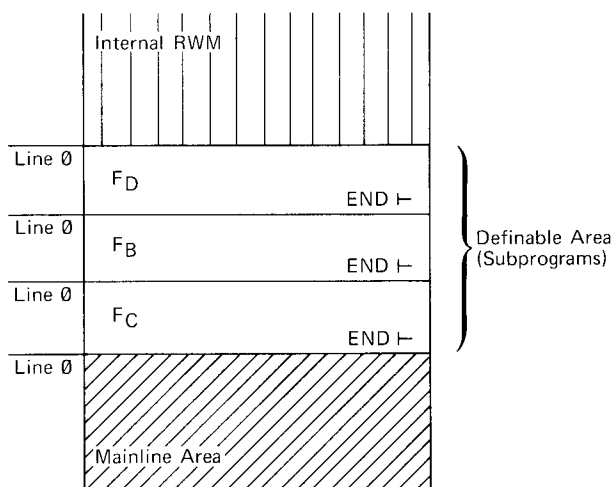


Figure 2-2. The Definable Area

If a new subprogram is now stored, it appears immediately below the last subprogram in the area (in the case shown in Figure 2-2, below F_C); the definable area expands and the mainline area moves down. The number of available R registers decreases accordingly.

If a subprogram is scratched (deleted), then the subprograms below it, and the mainline area, move up to fill the gap. The number of available R registers increases accordingly.

If a subprogram (say F_B) is not scratched and a new subprogram with the same name (F_B) is to be stored, then the new subprogram automatically replaces the old; that is, the new subprogram occupies the space in memory previously occupied by the old subprogram. Subprograms with a lower position in memory, and the mainline area, then automatically move up or down to compensate for any difference in length between the old and the new subprograms. (However, as will be seen in the next chapter, it is a good practice to first scratch a subprogram if a new one is to replace it.)

Chapter 3

WRITING SUBPROGRAMS

This chapter presents some rules for writing subprograms and includes some examples; at the end of the chapter, a summary of some of the syntaxes is presented. The examples are very simple because their purpose is to illustrate use of the

UDF Block. The simplicity of the examples does not mean that subprograms necessarily have to be short or simple; they can have as many lines, and can be as complicated, within the limits of the calculator, as the programmer pleases.

THE KEY MNEMONICS

The three overlays, and a table of the mnemonics associated with each key, are shown on the fold-out at the back of this manual. The table shows two groups of keys: the meanings of the 'control' keys are fixed; the meanings of the 'user definable' keys are determined by the subprograms which they are made to represent.

Throughout the remainder of this manual, the symbol 'F★' will be used to denote any, as opposed to a specific, user definable key. Thus F★ can represent any of 25 keys: F_A through F_E, G_A through G_J, and H_A through H_J. (There is no actual mnemonic 'F★' in the calculator.)

WHICH TYPE?

Before you start to write a subprogram it is important that you decide which type of subprogram (a subroutine, a function or a procedure) it is to be. The three types of subprogram each have special requirements. The choice of type of subprogram must be based not only on what the subprogram is to do, but also on how you want to use it (see the preface to this manual). There are few 'hard-and-fast' rules and in many cases a particular subprogram (one to calculate N-factorial, for example) could be easily rearranged to fit all three types. In general your subprogram should be:

required activity that you wish to execute easily from the keyboard.

a. A procedure – if it is some frequently

b. A subroutine – if it is some activity that you wish to call frequently from a program and that results in either no value (e.g. printing a message) or in many values.

c. A function – if it is a process that results in only one value and if that value is to be inserted into an arithmetic expression (in the same way, for example, as the value '2' is inserted in place of ' $\sqrt{4}$ ' into the expression ' $6\sqrt{4}/R1 \rightarrow A$ ').

GENERAL CONSIDERATIONS

Each subprogram is completely isolated from all other subprograms and from any mainline program in the memory. Thus, a subprogram can be written without regard to any other subprogram or to the mainline program, unless (and this applies in particular to the procedure subprogram) the subprogram is to include actual register names instead of dummy and local variables.

In each subprogram, the program lines are numbered consecutively and always start at line 0. Any branching statements in a subprogram can apply only to lines within that same subprogram; this does not apply to those statements used specifically to branch to other subprograms (see Chapter 4).

GENERAL CONSIDERATIONS

(cont'd)

Except in procedure subprograms, P-numbers can be used for all variables; that is, in place of values and register names. Known constants can, of course, be given their proper values — there is nothing to be gained by making the parameter lists passed to a subprogram longer than necessary. Ensure that, in each subprogram, the P-numbers are consecutive (that no number in the sequence is missing) and that P1 is included. Also ensure that the local variables have higher P-numbers than the dummy variables.

Do not attempt to write subprograms in such a way that the local variables in one correspond to

the local variables that have the same P-numbers in another (unless those P-numbers are to be passed in a parameter list). This sometimes can be done, but its use is very limited; it requires that no other subprogram with local variables be run as long as you wish the correspondence of the local variables between the original subprograms to last.

'END' must be the last statement in each subprogram. END in the definable area behaves differently from END in the mainline area — it is closer to being a RETURN statement rather than an 'END' statement (see 'The END Statement', at the end of this chapter).

THE PROCEDURE SUBPROGRAM

The main difference between a mainline program and a procedure subprogram is that the first statement in the subprogram must be IMMEDIATE EXECUTE (IEX); however, see 'Naming Subprograms', below. Once that subprogram is stored, only the key that represents it (F★) need be pressed to start that subprogram.

As an example, suppose that you frequently total lists of numbers and calculate the average value of each list. The following three subprograms, stored as procedures, would greatly simplify that process. The first subprogram enables you to press one key to initialize the list; the subprogram clears the two registers that are to be used (respectively) to count the numbers in the list, and to accumulate the total of the list.

```
0:
IEX ;0+R1+R2F
1:
END F
```

The second subprogram gives you an 'accumulate' key, which is pressed after each number in the list has been keyed; it increments a counter (R1) and adds the number just keyed into the display to the current total (in R2):

```
0:
IEX ;1+R1+R1F
1:
Z+R2+R2F
2:
END F
```

The above subprogram utilizes the 'implied Z' feature, which is why it uses the number just keyed. If no number was keyed, then it uses whatever number is currently in Z — which is not necessarily the number currently displayed.

The third subprogram then gives you your 'average' key; it calculates the average of the list:

```
0:
IEX ;DSP R2/R1F
1:
END F
```

(The procedure to store and run these subprograms is given in Chapter 4).

Once a procedure subprogram is stored, you never need to press any of the usual keys, such as GO

◆◆◆◆◆ THE PROCEDURE SUBPROGRAM ◆◆◆◆◆

TO, EXECUTE, or RUN PROGRAM, in order to start the subprogram.

A procedure subprogram can also contain ENTER statements; however, you must be careful not to

use this subprogram if you are currently halted in the ENTER mode for any other program or subprogram (see 'Nesting Enter Statements', later in this chapter).

◆◆◆◆◆ THE SUBROUTINE SUBPROGRAM ◆◆◆◆◆

The subroutine subprogram is similar in form to a mainline subroutine, except that the subprogram uses P-numbers. The subroutine subprogram is used in essentially the same way as a local subroutine; it is called and executed, and then a return is made to the calling program (to the next line after the one containing the 'CALL' statement — see Chapter 4).

Following is an example of a subroutine subprogram. It clears selected consecutive R registers by storing zeros into them. The registers to be cleared are specified, whenever the subprogram is called, by the two numbers in the parameter list. The first parameter (corresponding to P1) is the number of the first R register to be cleared, and the second parameter (corresponding to P2) is the number of the last R register to be cleared; the subprogram itself generates the intermediate register numbers, by incrementing (by one) the value originally assigned to P1 until it is equal to P2.

```
0:
0→RP1†
1:
IF P2>P1:P1+1→P1
;GTO 0†
2:
END F
```

As an example, the statement 'CALL F★ 40, 59' results in registers R40 through R59 (inclusive) being cleared. (Chapter 4 explains the use of 'CALL' and includes the procedure to store and run the example subroutine subprogram.)

A subroutine subprogram can contain ENTER statements.

◆◆◆◆◆ THE FUNCTION SUBPROGRAM ◆◆◆◆◆

A function subprogram, once stored, becomes a keyboard function (in the mathematical sense) and is used in exactly the same way as any other keyboard function (such as the square root), except that it can have a multi-parameter argument. The function subprogram is subject to the same arithmetic hierarchy as are other functions.

A function subprogram is written in the same way as a subroutine subprogram except that the final result is treated in a different manner. The result of a function subprogram has to be stored in a

special internal register (not otherwise available to the user) named 'F'. In the subprogram, that value which is to be the 'value of the function' must be stored by means of the 'goes to F' key (→ F)†. When the subprogram ends, the value last stored in F becomes the value to be substituted for the function in any arithmetic expression containing the function (and its argument).

†The mnemonic →F is obtained by means of the DEFINE/→F key; it cannot be obtained using the assignment key (→) and the F key.

◆◆◆◆◆ THE FUNCTION SUBPROGRAM ◆◆◆◆◆

(cont'd)

Following is an example function subprogram, to calculate N-factorial:

```
0:
0→P2;1→P3┐
1:
IF P2≠P1;P2+1→P2
;P2P3→P3;JMP 0┐
2:
P3→F┐
3:
END┐
```

The argument (N) for this function is designated in the same way as for any function — the function key followed by the argument (e.g., F★ 6). In this case, N corresponds to P1, while P2 and P3 are local variables. The final result of the function is stored into F in line 2. Notice that the result first had to be generated in (in this case) P3 because F cannot be manipulated in any way; the user has no control over F other than to store a value into it, by means of the 'goes to F' state-

ment in a function subprogram. The fact that, in the example, END follows immediately after the 'goes to F' statement is not intended to imply that this must always be so; there can be other statements between.

Unlike the square root function and the functions supplied by the 11221A Mathematics Block, function subprograms can have multi-parameter arguments. For example, the following subprogram, which calculates the square root of the sum of two squares, requires a two-parameter argument:

```
0:
√(P1P1+P2P2)→F┐
1:
END┐
```

When the argument of a function contains more than one parameter, the argument must be enclosed in parentheses; using the preceding example, 'F★ (3,4)' assigns 3 and 4 to P1 and P2 respectively; which, in this case, results in five being returned as the value of the function.

Function subprograms cannot contain ENTER statements.

◆◆◆◆◆ NAMING SUBPROGRAMS ◆◆◆◆◆

The 'name' of a subprogram is usually the mnemonic (F★) of the key that represents it. However, the mnemonic never appears in a listing of the subprogram. To facilitate identifying a subprogram both in the listing and in the display, it can be named by means of a literal (characters inside a quote field). For example, the N-factorial function subprogram might reasonably be named "N!"; the listing of the N! subprogram then appears as:

```
0:
"N!";0→P2;1→P3┐
1:
IF P2≠P1;P2+1→P2
;P2P3→P3;JMP 0┐
2:
P3→F┐
3:
END┐
```

The literal must be the first statement in the subprogram; then, whenever the key (F★) defined by that subprogram is pressed (or displayed, or printed), the name (e.g. N!, without the quotation marks) is automatically substituted for the normal mnemonic (F★). Even though the name can have up to 16 characters (see below), once it is stored in the subprogram, it requires no more memory space, when appearing as a key mnemonic in a statement in another program, than does the key 'F★'.

The literal used to name a subprogram can have as many characters as line length will allow; however, it is effectively limited to 16 characters because only the first 16 will ever be displayed, or printed, as the mnemonic for F★.

When writing a literal to name a subprogram, it is often useful to include a blank space as the last character in the name. The space serves to separ-

◆◆◆◆◆ NAMING SUBPROGRAMS ◆◆◆◆◆

ate the name from the next character (usually the start of a parameter list) in the display or in a program listing. Without that space, confusion sometimes arises as to which is the last character in the name and which the first in the parameter list.

If a procedure subprogram is to be named, then the literal must be the first statement, and the IMMEDIATE EXECUTE must be the second statement, of that subprogram. The main reason for naming a procedure is to more easily identify a listing of the subprogram.

◆◆◆◆◆ NESTING 'ENTER' STATEMENTS - A DON'T ◆◆◆◆◆

When using (and writing) subprograms, take great care to ensure that you do not nest 'ENTER' statements. When halted in the 'ENTER' mode, it is not possible to run any subprogram containing an 'ENTER' statement. If you do so, 'NOTE 11' will appear — if this occurs, press no other keys, but carefully follow this procedure:

3. Press CLEAR RETURN EXECUTE.

The procedure will also set the program line counter to line zero in the mainline program; however, it is the only sure way to clear the nested 'ENTER' statements (without pressing ERASE).

1. Press CLEAR.
2. Press STOP as often as necessary until the mnemonic 'STP' appears in the display.

◆◆◆◆◆ SYNTAXES - A SUMMARY ◆◆◆◆◆

P-NUMBERS

P<quantity> or P (<expression>)
P2 PR14 P(P2+1)

P-numbers designate both dummy and local variables. Parameters correspond to dummy variables, but not to local variables, whenever the subprogram is used.

P-numbers are used in subroutine and function subprograms, but not in procedure subprograms.

P-numbers may be used in any order in a subprogram; however, the numbers used should be consecutive and should include P1 (P0 does not exist).

If a dummy variable is assigned a register name (A, R1, RR16, etc.) as its parameter, then that actual register is used during execution of the

subprogram. If the value in that register is changed during execution, then it retains that changed value afterwards.

If a dummy variable is assigned a value other than a register name, then that value is stored in a part of memory that is not otherwise available to the user.

Local variables have working registers established for them, whenever the subprogram is executed, immediately below the programming in the mainline area. The number of working registers that will be established can be determined by subtracting the number of parameters passed (i.e., the number of dummy variables) from the number of the highest P-number used in that subprogram. The working registers use the same area as the highest numbered (available) R registers; however, working registers rarely coincide with specific R registers.

 SYNTAXES - A SUMMARY 

THE VALUE OF A FUNCTION

```
<value>→F ('goes to F')  
P2→F ; 6P2/P3→F ;
```

Used only in a function subprogram. Designates the result of the subprogram; that is, that value that is to be returned as the 'value of the function'. If there is more than one 'goes to F' statement in the subprogram, only the last one executed designates the final value of the function. When function subprograms are nested, or when more than one function is used in an expression, the calculator automatically distinguishes between the final values of the different functions.

If the DEFINE/→F key is used as the first key in a statement, it has its DEFINE meaning; otherwise, it has its 'goes to F' meaning.

THE 'IMMEDIATE EXECUTE' STATEMENT

```
IEX ;
```

IMMEDIATE EXECUTE is used only in procedure subprograms. It must be the first statement in the subprogram, unless the subprogram is to be named (see 'Naming Subprograms'), in which case IMMEDIATE EXECUTE must be the second statement.

IMMEDIATE EXECUTE causes the subprogram to be started as soon as its key (F★) is pressed.

(See also 'The Immediate Execute Mode', in Chapter 5.)

THE 'END' STATEMENT

```
END ↵
```

Every subprogram should have END as its last, and only as its last, statement.

'END' in the definable area of memory does not have the same properties that it has in the mainline area (clearing flags and return addresses, etc.). In the definable area END acts as follows:

1. When END is executed from the keyboard, or when STORE is pressed to store an 'END' statement, the program line counter goes to line zero in the mainline area.
2. When any type of subprogram is executed from the keyboard, END causes program execution to cease and returns the program line counter to line zero in the mainline area.
3. When a subroutine subprogram is executed under program control, the END statement acts exactly like a RETURN statement in the mainline area — it causes a return to the program line following the line containing the subroutine subprogram call.
4. In a function subprogram the END causes the last value stored into F (see 'The Value of a Function' in this chapter) to be returned as the value of that function. The calculator then automatically continues to complete the statement that caused the function subprogram to be executed.

THE 'RETURN' STATEMENT

```
RET ↵
```

In a subprogram, a 'RETURN' statement which has no matching 'GO TO SUB' statement acts like the 'END' statement, described above. The exception is that when RETURN is stored, the program line counter goes to the next line in the subprogram, and not back to the mainline area.

Chapter 4

MECHANICS OF OPERATION

This chapter describes the mechanics associated with using subprograms: establishing the place in memory where a particular subprogram is to start; storing the subprogram; editing it; using it; and, finally, recording it.

As in Chapter 3, 'F★' is used to denote any key (as opposed to a specific key) which can be defined by a subprogram. Thus F★ can represent any of twenty-five keys: F_A through F_E, G_A through G_J, and H_A through H_J.

ESTABLISHING A HEADING

Each subprogram has a 'heading' which indicates (to the calculator) what type of subprogram it is — a subroutine/procedure or a function — and the key, F★, which represents it. The heading locates the starting point of the subprogram in the definable area. The 'GO TO' statement is used to establish the headings for procedure subprograms and for subroutine subprograms. The 'DEFINE' statement establishes the headings for function subprograms only.

line zero for that subprogram. Once a heading is established, it remains established until it is 'scratched' (see also 'The Scratch Statement' and 'Accessing a Subprogram' below).

THE SUBROUTINE/PROCEDURE

Press GO TO F★ EXECUTE (where F★ is to be a subroutine or a procedure subprogram).

This establishes the heading (if it does not already exist) for either a subroutine or procedure subprogram. It also sets the program line counter to

THE FUNCTION

Press DEFINE F★ EXECUTE (where F★ is to be a function subprogram).

DEFINE F★ is similar to GO TO F★ (above) except that it establishes the heading for a function subprogram.

The mnemonic 'DEF' (define) is obtained by pressing the DEFINE/→ F key as the first key in the statement, otherwise the key has its 'goes to F' meaning.

SUBPROGRAM STORAGE

ACCESSING A SUBPROGRAM

Press GO TO F★ EXECUTE (where F★ can be the name of any type of subprogram).

Executed from the keyboard, sets the program line counter to line zero in the subprogram F★ (if already established). Subsequent activity then depends upon the next key pressed.

GO TO F★ is not normally used within a program. It can be used but the result will depend upon the circumstances (e.g., the END in the subprogram would cause NOTE 07 to appear, and the 'goes to F' would cause NOTE 02).

STORING A SUBPROGRAM

First establish the heading by 'GO TO F★' for a subroutine or procedure subprogram, or by 'DEFINE F★' for a function subprogram (these statements also set the program line counter to line zero for that subprogram). Then key and store the lines of the subprogram in exactly the same way as the lines of a mainline program are keyed and stored. Branching statements apply to lines only within that subprogram, until the program line counter has returned to the mainline area.

When STORE is pressed to store an 'END' statement, the line containing the 'END' is not seen;

SUBPROGRAM STORAGE

STORING A SUBPROGRAM (cont'd)

only the 'end-of-line' symbol appears, indicating that the program line counter has returned to the mainline area, to line zero.

LISTING A SUBPROGRAM

GO TO F★ LIST causes the printer to make a listing of the subprogram (F★). When the listing is completed the program line counter returns to

line zero in the mainline area.

Attempting to list a subprogram whose heading has not yet been established results in a heading being established and an 'END' statement being stored as the subprogram. This will use approximately the equivalent of one R-register. (With a completely empty definable area, if you listed all twenty-five definable keys, you would use up the equivalent of twenty-five R-registers.)

EDITING SUBPROGRAMS

RECALLING AND MODIFYING

Once the program line counter is set to a line within a subprogram, lines can be recalled and modified, deleted or inserted, exactly as in the mainline area. Any 'GO TO' statements executed from the keyboard must be to line numbers within the subprogram currently being edited.

Any of the following operations returns the program line counter to line zero in the mainline area:

1. Storing or executing an 'END' statement.
2. Pressing RECALL when the line containing the 'END' statement is currently displayed.
3. Stepping, by means of the FORWARD key, past the 'END' statement.
4. Executing a RETURN when there was no matching GO TO SUB executed while in the subprogram.

THE 'SCRATCH' STATEMENT

(MEMORY ERASE automatically erases all subprograms as well as all mainline programs and stored data.)

Press SCRATCH F★ EXECUTE
or SCRATCH F★₁, F★₂ EXECUTE

SCRATCH is used to selectively scratch (delete) one or more subprograms. Once a subprogram (F★) is scratched, its heading is no longer contained in memory.

Subprograms can be scratched only from the keyboard; an attempt to scratch from a program results in NOTE 24 appearing, indicating an error during execution.

See also 'Recording Subprograms' in this chapter.

RUNNING SUBPROGRAMS

(When loading the example subprograms given below, refer if necessary to the foldout, at the back of this manual, which shows the mnemonics obtained when the keys associated with the UDF Block are pressed.)

Once a subprogram is stored it is very straightforward to use:

The Procedure Subprogram is started by pressing the key (F★) which it defines.

The Subroutine Subprogram is started by means of a 'CALL' statement (explained below).

The Function Subprogram is used when a statement containing its name and argument is executed.

USING PROCEDURE SUBPROGRAMS

A procedure subprogram is started by pressing its key, F★. Depending upon how the subprogram is

◆◆◆◆◆ RUNNING SUBPROGRAMS ◆◆◆◆◆

written, data may have to be keyed either before it is run or during an 'ENTER' statement, if it has one. Halts for data entries require RUN PROGRAM to be pressed, in the normal way, in order to resume subprogram execution.

As an example of procedure subprograms, load and run the three procedures (described in Chapter 3, under the heading 'The Procedure Subprogram') which are used to total a list of numbers and to calculate its average value. The following process loads the three procedures as F_A , F_B , and F_C , respectively.

(Press ERASE)

1. Press GO TO F_A EXECUTE, and store the following program lines:

```
0:
  IEX ;0+R1+R2+
  1:
  END F
```

2. Press GO TO F_B EXECUTE, and store the following program lines:

```
0:
  IEX ;1+R1+R1+
  1:
  Z+R2+R2+
  2:
  END F
```

3. Press GO TO F_C EXECUTE, and store the following program lines:

```
0:
  IEX ;DSP R2/R1+
  1:
  END F
```

With the above three subprograms stored, a list of numbers can be totalled and averaged as follows:

1. Press F_A ('initialize').
2. Key each number in the list and press F_B ('accumulate') after each number.
3. Press F_C ('average') — the average of the list appears in the display.

THE 'CALL' STATEMENT

```
CALL<Subroutine Subprogram Name> [<List>] +
CALL F★<Parameter> [,<parameter>...]
```

CALL is used to branch to a subroutine subprogram. The 'CALL' statement must be the last statement in a line. A parameter list is not required if the subprogram called has no dummy variables — a subprogram to print a message, for example. When execution of the subprogram is completed, a return is made to the calling program (or calling subprogram) — to the line following the line which contains the 'CALL' statement. If the call is executed from the keyboard, then program execution ceases and the program line counter is set to line zero in the mainline area.

USING SUBROUTINE SUBPROGRAMS

Following is the procedure to store and call the example subroutine subprogram shown under the heading 'The Subroutine Subprogram', in Chapter 3; this subprogram can be used to clear any number of (consecutive) R-registers. Load the subprogram as F_D .

Press GO TO F_D EXECUTE and store the following program lines:

```
0:
  0+RP1+
  1:
  IF P2>P1;P1+1+P1
  ;GTO 0+
  2:
  END F
```

The following 'CALL' statement will call F_D and clear registers R40 through R59, inclusive:

```
CALL FD 40,59+
```

The 'CALL' statement can be executed from the keyboard — press EXECUTE — or stored as the last statement in a line, of either a mainline program or of another subprogram.

USING FUNCTION SUBPROGRAMS

A function subprogram is used in an expression by pressing its key, $F★$, followed by its argument; if the argument is negative, or includes more than one parameter, or is an expression, then it must

RUNNING SUBPROGRAMS

USING FUNCTION SUBPROGRAMS (cont'd)

be enclosed in parentheses. When the expression containing the function is executed, execution of the expression is temporarily suspended while the function is executed; when the value of the function is returned, execution of the (suspended) expression is continued.

Following is the procedure to store and run the N-factorial function (see 'The Function Subprogram' in Chapter 3), as F_E :

Press $DEFINE F_E EXECUTE$ and store the following program lines (the exclamation in the literal is obtained by pressing the STOP key):

```
0:
  "N!" 10→P2 11→P3 1
1:
  IF P2≠P1 ; P2+1→P2
  ; P2P3→P3 ; JMP 0 1
2:
  P3→F 1
3:
  END 1
```

The following are example statements illustrating use of the function subprogram (the mnemonic 'N!' will appear, not ' F_E ', because the first statement in the subprogram is a literal):

```
FE 6 ;
```

calculates $6!$ (=720).

```
16XXX/FE 3→C ;
```

calculates the expression $16x^3/3!$ and stores the result in C (the actual value stored into C depends upon the value currently stored in register X).

```
FE FE 3 ;
```

calculates $(3!)!$ (=6!=720).

All of the above statements can be executed from the keyboard — press $EXECUTE$ — or stored as statements in a program.

RECORDING SUBPROGRAMS

This discussion describes the recording and loading of subprograms on magnetic cards. The information given in the calculator's Operating and Programming Manual, describing the general aspects of recording and loading, still applies.

GENERAL CONSIDERATIONS

A record can be made either of one subprogram or of a series of subprograms. The advantage of single-subprogram records is that they enable any combination of subprograms to be loaded, in any order. The 'series' record, on the other hand, has the advantage that it enables a specific combination of subprograms to be loaded very rapidly.

Recording always starts at the current setting of the program line counter and continues until all programming below that point has been recorded. Loading works in essentially the same way. It is not possible to record just subprograms — mainline programming, even though it might be only

the null program, will also be included on the record. The mainline programming currently in the memory will always be lost whenever a record of one or more subprograms is loaded. In general, mainline programs should have their own records, although in some cases it is advantageous to have both subprograms and mainline programs on the same record.

If a mainline program does not require the UDF Block, then it is a good rule to record it when that block is not installed. This does not apply if you can be sure that the UDF Block will always be installed in the calculator. The reason is, that if such a program is recorded with the block installed, then that record cannot be properly loaded into a calculator which does not have the block installed. On the other hand, a record made when the UDF Block is not installed, can be properly loaded into a calculator which does have the block.

RECORDING SUBPROGRAMS

SINGLE SUBPROGRAMS

When only one subprogram is to be recorded, it should be the only subprogram in memory. Also, there should be no mainline programming, other than the null program. When the record is made, it includes the first statement in the subprogram but it does not include the subprogram's heading (see 'Establishing A Heading' at the beginning of this chapter); the heading specifies which key ($F\star_1$) represents that subprogram. Without the heading, there is nothing on the record to specify which key represents that subprogram; therefore, the user can specify any key ($F\star_2$) he pleases when he loads that record back into memory.

The initializing process for both making and loading the record of a single subprogram depends upon the type of subprogram:

- a. If it is a subroutine or a procedure, press GO TO $F\star$ EXECUTE.
- b. If it is a function, press DEFINE $F\star$ EXECUTE.

Then start the recording or loading in the usual way, by pressing either RECORD EXECUTE or LOAD EXECUTE, as appropriate. Be sure to mark on the record whether a 'GO TO' statement or a 'DEFINE' statement is to be used to initialize loading; the subprogram will not run properly if the wrong statement is used (scratch the wrong subprogram and reload it).

Before loading a subprogram always ensure that the key ($F\star$) that you are about to use does not already represent a subprogram currently in memory. Either press ERASE or, when that is not desirable, execute a 'SCRATCH' statement (SCRATCH $F\star$) to scratch the current subprogram from the memory.

When single subprograms are loaded they are stored in the definable area of memory in the same order as the order in which they are loaded (see Figure 2-2 in Chapter 2).

A SERIES OF SUBPROGRAMS

This type of record is a complete 'map' of all programming in memory at the time the record was made. Unlike the single-subprogram record, it

does include the heading (i.e., the specific key, $F\star$) for each subprogram. This type of record can, if required, include the necessary mainline programming (that is, other than just the null program).

Before recording, ensure that only the required programming is in the memory. To initialize both the record and load processes, press:

GO TO SCRATCH EXECUTE

Then record or load in the usual way, by pressing RECORD EXECUTE or LOAD EXECUTE, as appropriate.

When loading, the record automatically replaces all programming in memory; it is not necessary to first press ERASE.

LOADING UNDER PROGRAM-CONTROL

The 'LOAD' statement can be executed as a statement in a program; however, from a practical point of view, its use is limited (as far as loading subprograms is concerned) to loading subprograms which have been recorded by means of the 'GO TO SCRATCH' statement. When the 'LOAD' statement is encountered, program execution is suspended while the programming on the record completely replaces all programming currently in memory. Program execution then resumes automatically at line zero in the mainline area.

The programmed 'load', which must be only in a mainline program, requires two statements, 'GO TO SUB SCRATCH' and 'LOAD', used as follows:

```
GSB SCR ;LOAD F
```

The two statements must be on the same line and 'LOAD' must be the last statement on the line.

Do not use 'GO TO SCRATCH' in place of 'GO TO SUB SCRATCH'. If you do, then program execution will resume at the beginning of the first subprogram instead of at line zero in the mainline area.

DIAGNOSTIC NOTES

Most of the execution and syntax errors associated with the UDF Block are similar in form to those of the basic calculator and cause the same 'notes' to appear. The UDF Block adds only one new note — NOTE 24.

Following is a brief description of the most likely errors (not all possible errors are listed). The symbol † implies either the STORE or EXECUTE instructions.

- NOTE 01
1. 'CALL' statement not the last statement on the line.
 2. Key other than F★ or IEX follows SCR.
 3. Key other than 'comma', 'semicolon' or † following F★ in a 'SCRATCH' statement.
 4. Any other key in an 'IEX' statement (other than SCR IEX — see 'The Immediate Execute Mode', in Chapter 5).
 5. CLL not the first key in a statement (also use of CLL during a halt for an 'ENTER' statement).
 6. Any key following CLL except F★ or a label enclosed in quote marks. (A 'label in quote marks' is the same as the line label in the mainline area; it is not the same as the literal that is the first statement in a subprogram — in a 'CALL' statement, the former has to be keyed character by character while the latter appears as soon as the key F★ is pressed.)
 7. Key other than 'semicolon' or † after 'DEFINE F★'.
 8. More than one F★ used in a 'GO TO' or 'GO TO SUB' statement.
- NOTE 02
1. Executing a 'goes to F' statement when the function was started by means of a 'GO TO' statement.
 2. Executing a P-number as a value in an expression when the subprogram was called incorrectly (e.g., by a 'GO TO' statement), or the P-number was included in a procedure subprogram.
- NOTE 05
- Attempt to use a P-number whose parameter is zero or less than zero [P0, P(-1), etc.].
- NOTE 06
- Attempt to store into a 'local register' (Pn) when the subprogram was called incorrectly (e.g., by a 'GO TO' statement) or when the P-number is used in a procedure subprogram.
- NOTE 07
- Executing a RET or an END (only in the definable area) without a matching GSB, CLL, or function subprogram call.
- NOTE 09
- See Chapter 5.
- NOTE 11
- See 'Nesting Enter Statements' in Chapter 3.
- NOTE 24
- Attempting to execute a 'SCRATCH F★' during program execution.

Chapter 5

ADDITIONAL INFORMATION

This chapter contains information, about the UDF Block, that will be needed only occasionally. Mostly it will be needed to enable the user to

understand certain occurrences, such as the 'immediate execute' mode, which might otherwise be a source of confusion.

◆◆◆◆◆ THE 'IMMEDIATE EXECUTE' MODE ◆◆◆◆◆

The immediate execute mode is established any time an 'IEX' statement is executed, except when it is executed in the usual way in a procedure subprogram. Once the mode is established, all subprograms, except function subprograms, automatically become 'immediately executable', even though they do not contain an 'IEX' statement.

Even though the mode is established, all subprograms can still be used exactly as before. However, a subroutine subprogram is immediately executed if its key (F★) is pressed as the first key in a statement. When started in this way, the subroutine subprogram will not run properly if it contains P-numbers (either NOTE 02 or NOTE 06 appears). On the other hand, if the subroutine does not use P-numbers, but instead uses actual values and register names, then it will run properly (notice that such a subroutine subpro-

gram is actually a procedure subprogram without the 'IEX' statement).

The mode can be established in any one of three ways:

1. Executing an 'IEX' statement from the keyboard.
2. Executing an 'IEX' statement in a mainline program or in any subprogram (except in the usual way in a procedure subprogram).
3. Executing a procedure subprogram as if it were a subroutine subprogram (i.e., by means of a 'CALL F★' statement).

A 'SCRATCH' statement is used to remove the 'immediate execute' mode:

```
SCR IEX F
```

◆◆◆◆◆ 'CALLING' LOCAL SUBROUTINES ◆◆◆◆◆

When the UDF Block is installed, local subroutines (in the mainline area) can be written using dummy and local variables (P-numbers) and can then be called by means of a 'CALL' statement with parameters. In general, the rules applicable to calling subroutine subprograms apply to these local subroutines except that the call must be by means

of a literal — a name inside a quote field. The same literal must be the first statement in the subroutine and RETURN must be the last statement. It is not possible to branch to a local subroutine by means of its line number when the 'CALL' statement is used.

◆◆◆◆◆ CAUSES OF NOTE 09 ◆◆◆◆◆

Several things cause NOTE 09, but it is often difficult to trace the actual cause because too many variable factors are involved. The one obvious cause is writing or storing too long a line.

NOTE 09 can be caused by nesting subroutines and

subprograms too deeply, but it is often not possible to predict beforehand what constitutes 'too deeply'. Without the UDF Block and given 'ideal' circumstances (nothing else affecting the depth), local subroutines can be nested thirty-one deep. When the UDF Block is installed this depth is

CAUSES OF NOTE 09

essentially doubled, but, as subprograms can also be nested, the depths that local subroutines can be nested depends (in part) on how many subprograms are nested. By themselves subroutine subprograms can be nested as deeply as can local subroutines (except that no more than 25 subprograms can be stored at one time). Function subprograms, on the other hand, cannot be nested more than four deep.

Additional factors which affect the nesting include: the type of subprograms currently being nested; the nature and length of any expression whose execution has been temporarily suspended to allow a function subprogram, within that expression, to be executed; the number of parameters currently being passed; and the nature of the current parameters, whether they are register names, numbers, or expressions (expressions which might themselves contain defined functions!).

It is also possible, without exceeding line length, to have too many parameters in a 'CALL' statement and so cause NOTE 09 to appear during execution. Again, what may be too many parameters in one situation may be perfectly acceptable in another situation. The following examples illustrate this.

The maximum possible number of parameters in any given call depends upon the types of parameters used and is limited by line length. Numbers with many digits obviously require more keystrokes than do, say, two-digit register names. The largest number of parameters, twenty-five, can be passed if they are all one-digit, unsigned integers:

```
0:
CALL FA 0,1,2,3,4
,5,6,7,8,9,0,1,2
,3,4,5,6,7,8,9,0
,1,2,3,4F
```

Even though this call has twenty-five parameters, it is actually the 'worst case' situation because none of the parameters are register names. As each

number is used, it will be stored in the internal read-write-memory, in the same area shared by the 'nesting' information; thus, the length of the call can also affect the depth of nesting.

The following subprogram cannot be called by the above 'CALL' because it cannot accept that many parameters. Instead, it starts printing meaningless numbers and then stops with NOTE 09 displayed. If the parameter list is shortened to seventeen parameters, then the subprogram will operate correctly.

```
0:
PRT P1,P2,P3,P4,
P5,P6,P7,P8,P9,P
10F
1:
PRT P11,P12,P13,
P14,P15,P16,P17,
P18,P19,P20F
2:
PRT P21,P22,P23,
P24,P25F
3:
END F
```

If the same subprogram is written to generate its own P-numbers then it can be passed all twenty-five parameters, and it will accept all of them:

```
0:
1→XF
1:
IF 26>X;PRT PX;1
+X→X;JMP 0F
2:
END F
```


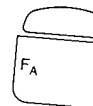



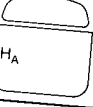
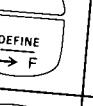

The above represent extreme situations; in general, it is fair to say that line length will most often be the deciding factor in determining the maximum number of parameters in a 'CALL' statement, and that there will usually be sufficient nesting-depth for most purposes.

- Note 1. 'b' represents a blank space.
- Note 2. Used as first key in a statement has 'DEFINE' meaning; otherwise has '→F' meaning.
- Note 3. Other USER DEFINABLE keys (F_B, G_B, etc.) have similar mnemonics (FB, GB, etc.).

The mnemonic for any of these keys is changed if the subprogram which it represents has a literal as its first statement (see 'Naming Subprograms' in Chapter 3).

APPENDIX


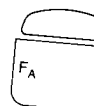
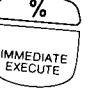
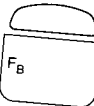
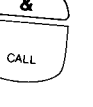
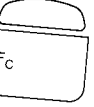
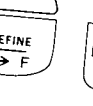

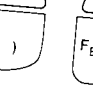
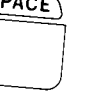
KEY MNEMONICS

CONTROL KEYS		USER DEFINABLE KEYS	
KEY	MNEMONIC	KEY	MNEMONIC
	SCR b (Note 1)		F A b (Note 1&3)
	IEX b (Note 1)		G A b (Note 1&3)
	CLL b (Note 1)		H A b (Note 1&3)
	DEF b or →F b (Note 1&2)	—	—
	P	—	—

KEY OVERLAYS

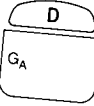
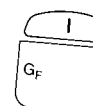
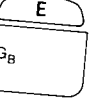
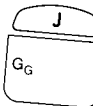


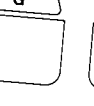
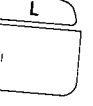


①

USER DEFINABLE FUNCTIONS I
11222A

②

USER DEFINABLE FUNCTIONS II (LEFT)
11222A

③

USER DEFINABLE FUNCTIONS III (RIGHT)
11222A

