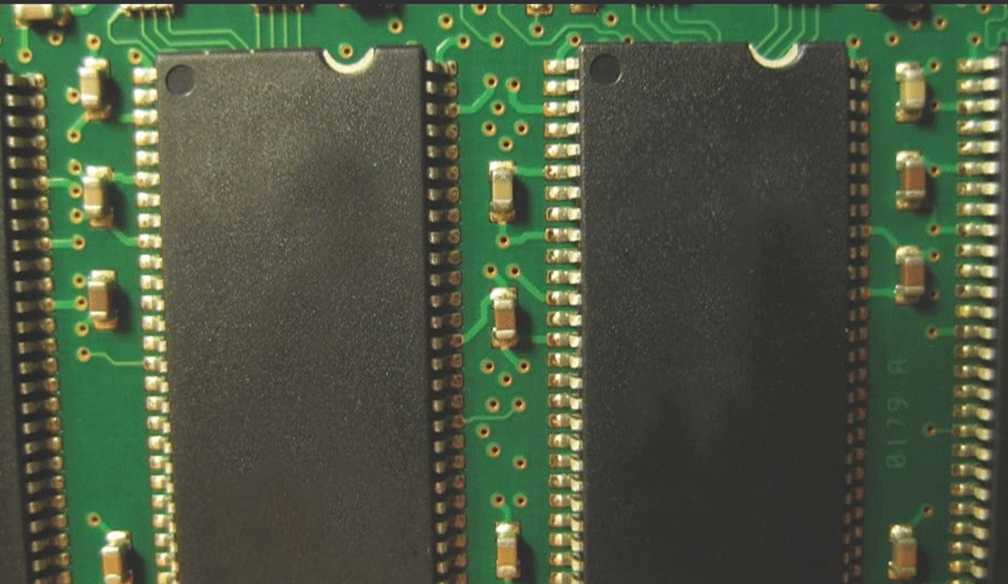
HP48 Machine Language

A Journey to the Center of the HP48 s/sx

Paul Courbis & Sébastien Lalande



HP 48S/SX Machine Language

Journey to the Center of the HP 48

by Paul Courbis and Sébastien Lalande

translated to English from the French

by Douglas R. Cannon

HP 48
Machine Language

Journey to the Center of the HP 48

by Paul Courbis and Sébastien Lalande

translated to English from the French

by Douglas R. Cannon

Grapevine Publications, Inc.

P.O. Box 2449

Corvallis, Oregon 97339-2449 U.S.A.

Acknowledgments

Printed in the United States of America

First Printing – December, 1993

We would like to give special thanks to:

Our respective families for the help and support they have given to us; Douglas R. Cannon for the enthusiasm and care with which he has translated this work; Marc Bernard de Courville for his numerous critiques; Ray Depew, without whom this edition would have never seen the light of day; Christophe Dupont de Dinechin for his program mSOLVER and his excellent remarks; Dominique Moisescu for his program, SSAG; Christophe Nguyen for his programs CIRCLE and BANNER; Yann Rousse; Jean Tourrilhes; the Maubert Electronic Company; all the members of the `comp.sys.hp48` group; and all those who have contributed with their remarks and ideas for the realization of this work.

Note to the Reader

This work has been designed for both the beginner and the advanced programmer. It contains information on the "classical" uses of the HP 48 as well as methods of accessing resources that are not documented by Hewlett-Packard.

The book is divided into four parts:

- **Part One** is to help you become familiar with the basic applications of the HP 48. Among these are: reverse polish notation, the stack, and the standard programming language. Also included are exercises that we suggest you use to help you understand these principles.
- **Part Two** will teach you the hidden resources of the HP 48 in a manner that is clear and helpful for a programmer of any level. This initiation course in machine language can later serve as an excellent reference manual.
- **Part Three** is a library of various programs that are ready to use. There are games, mathematical programs, utilities, music and more.
- The Appendices in the last part contain programming references (an exhaustive list of error messages, a complete list of instructions, etc.).

Important Note: *The different versions of the HP 48 (S and SX) are taken into account in this work: All programs, diagrams and other information (with the exception of the plug-in cards) are independent of the type of machine you have.*

Now it's up to you! We hope you enjoy the reading.

Table of Contents

Part One: The HP 48

The basic principles of HP 48 usage, as described by the manufacturer.

Part Two: Machine Language

The HP 48's hidden resources:
How to do more than Hewlett-Packard intended.

Part Three: Library of Programs

A collection of useful, ready-to-use programs.

Mathematical Programs

Games

Miscellaneous Programs

Appendices

Answers to exercises; programmer's reference; glossary; index.

Part One:

The HP 48

Introduction

You have in your hands one of the best calculators on the market—if not indeed *the* best. Compared to other calculators, it is much more complex in functionality, yet much simpler to use, and capable of solving problems of great complexity.

Considering its vast assortment of internal functions and their power , the HP 48 system had to be powerful and yet usable by everyone, whether a skilled mathematician, an excellent programmer , a physicist, a statistician, or even someone who has nothing to do with these areas at all.

Since the capabilities of this machine are much dif ferent than those of a regular calculator, it often appears at first to be very complicated, when actually it is the simplest system there is. It is just a question of habit, and in a few days (with a little practice) you will master the HP 48.

The chapters of **Part One** cover a general vision of the standard use of the machine: a few tricks to learn, how to make simple programs, how to stay organized, etc. The goal of **Part One** is not to replace the Hewlett-Packard instruction manuals, but rather to show you the capabilities of your machine in a way that will make it easier to use those manuals.

The Hewlett-Packard manuals show many things that the HP 48 can do. With machine language, however, it is possible to access new resources and create programs that are much faster . That is what **Part Two** teaches you: With elegant examples accessible to programmers of all levels, it shows you what programming in machine language is like, and it also describes the internal structure of the HP 48. So even if you know nothing at all about machine language or assembly language, here is a good chance to learn!

Before we get to that, however, it is a good idea to know the normal uses of your machine. To aid you in your learning, there are program examples, ranging from elementary to very complex, found in **Part Three** (**Library of Programs**). By using these programs or modifying them as you wish, you will soon be able to write sophisticated programs.

1. First Approach to the HP 48

Your machine sits before your eyes, covered with buttons. The blue, orange, and white inscriptions don't seem to mean much at the first glance. But this should not alarm you. It is just like a Christmas tree: at first glance it looks like chaos, but if you take a moment to look at it, you notice that each decoration was placed carefully. It then becomes obvious that the creator was working thoughtfully.

Like every electrical appliance, the HP 48 needs current. Verify that the three batteries, in the back of the machine at the base, are in place and facing in the correct directions. The batteries on top and bottom should have the + side pointing left; the middle battery should point to the right.

The Keyboard

Next, turn it on. Simply press the [ON] button which is the lower left-most button (written in white).

Above this you will find two buttons [↱] (blue) and [↰] (orange). If you press any key by itself, the function written in white will be executed. Pressing the [↱] (blue) shift key first will cause the function in blue to be executed. Likewise, pressing the [↰] (orange) shift key first will cause the function in orange to the executed. For example, if you press [↱] first, the [STO] key then becomes the [RCL] key; you are actually pressing [↱][RCL], thus executing the command RCL, which we will later see stands for recall (to recall the contents of a variable).

Above the [↰] key is the [α] key. If you press [α] once, this activates alpha mode for one keystroke. Notice that some keys have a white letter to the right. If one of these is pressed after the [α] key, then that letter will appear on the screen. For example, pressing [α] then [SIN] gives the letter S, whereas pressing [SIN] by itself simply executes the sine function.

To remain in alpha mode for more than one keystroke, you must press [α] twice. To exit this mode, simply press [α] once more. To type 'AB' you would press the buttons: ['][α][α][A][B][ENTER].

The Screen

The screen is divided into 3 parts:

- Above the horizontal bar you will find the current status of the machine. This will always include the directory path between curly brackets ({}) (see **Chapter 3** for more on this subject). It may also include small numbers (**1**, **2**, **3**, **4**, and **5**) indicating the state of certain flags of the machine, an angle mode indicator (**RAD**, for "radians," or **GRAD**, for "gradians," or nothing for "degrees"), or the date and time.
- Below this, separated from the first section by a horizontal bar , are 4 lines:

  4:
  3:
  2:
  1:

  This is the stack (see **Chapter 2**).
- The third section, at the bottom of the screen, shows the current "menu" or "directory." This consists of six labels, each containing the name of a function or variable. Pressing the key directly below a label will execute that particular function. For example, the [A] key would execute the function shown in the first label of the menu, found in the lower left corner of the screen.

  Some labels have a small horizontal bar on top, which makes them look like little folders. These represent sub-menus or sub-directories. (**Chapter 3** covers menus and directories more thoroughly.) For example, if you were to execute the MEMORY command (press [←][MEMORY]), you would be placed in the memory menu:

  MEM | BYTES | VARS | ORDER | PATH | CRDIR

  You could then execute the VARS command (for example) by pressing the [C] key.

Exercises

1-1. What sequence of buttons would you need to press to get an =?

1-2. What sequence of buttons would you need to press to execute the function RCL?

2. Reverse Polish Notation

The HP 48 uses a calculating method called "Reverse Polish Notation" (RPN). To understand this notation, we must first define the principle of the stack.

The Stack

Imagine a stack of plates where the only accessible plate is the one on the top of the stack. The HP 48 temporarily stores objects in the same manner. The first four stack entries can be seen on the screen preceded by their stack number (1:, 2:, 3: and 4:). Obviously this doesn't look exactly like our stack of plates, since the first "plate" is on the bottom, but the principle is the same.

Although only the object at level 1 is available for use, there are commands that permit us to change the order of the stack. Before learning this, however, let's find out how to place objects on this stack.

The HP 48 handles many types of objects (real numbers, binary integers, strings, names, programs, equations, graphic objects, etc.). Each of these object types may be placed on the stack. To do this, simply type in the object and press [ENTER]. For example, to place the real number 123 on the stack, simply press the keys: [1][2][3][ENTER].

You then see the following on the screen:

```
4:
3:
2:
1:                   123
```

This signifies that the stack contains one object, 123, in level 1.

Note: The HP 48 will show only the first 4 stack entries, although the stack may contain many more. The size of the stack is limited only by the available memory.

Calculating in RPN

The different functions of the HP 48 (addition, subtraction, etc.) take their arguments from the stack. After the calculation, the result is placed on the stack.

Reverse Polish Notation is often difficult for those who are used to a standard notation. With continued use, however, you will find that RPN performs much better. In particular, RPN does away with parenthesis because the stack can store the intermediate arguments. For example, to calculate (2+3)(4+5), we would perform the following commands:

- Begin with an empty stack (if the stack isn't empty, use the CLR command—[→][CLR]—to clear it). The screen should look like this:

```
4:
3:
2:
1:
```

- Pressing [2][ENTER] shows:

```
4:
3:
2:
1:                                2
```

- [3][ENTER] shows:

```
4:
3:
2:                                2
1:                                3
```

Note that the 3 pushed the 2 to the second level of the stack. This is correct, since the "top plate" is now the 3.

- [+] adds the two numbers:

```
4:
3:
2:
1:                    5
```

- [4] [ENTER] shows:

```
4:
3:
2:                    5
1:                    4
```

- [5] [ENTER] shows:

```
4:
3:                    5
2:                    4
1:                    5
```

- [+] gives:

```
4:
3:
2:                    5
1:                    9
```

- And finally, [×] gives the result:

```
4:
3:
2:
1:                   45
```

We typed no parentheses, yet we were able to handle the intermediate results (5 and 9). Remember, a command takes its arguments (however many it needs) from the stack and places the result(s) onto the stack.

Managing the Stack

We have seen that various commands use only the first few stack entries, so how can the others be accessed? We have at our disposal commands to manage the stack. In particular, we can use the following commands:

- SWAP ([←][SWAP]) exchanges the stack entries in levels 1 and 2. For example:

```
4:
3:
2:                              2
1:                              1
```

After [←][SWAP]:

```
4:
3:
2:                              1
1:                              2
```

- DROP ([←][DROP]) drops (erases) the object in level 1. For example:

```
4:
3:                              3
2:                              2
1:                              1
```

After [←][DROP]:

```
4:
3:
2:                              3
1:                              2
```

- CLR ([→][CLR]) clears the stack. With the above stack, [→][CLR] would give:

```
4:
3:
2:
1:
```

These are the most common commands, but there are others. They can be accessed from the STK menu, which is in the PRG menu (press [PRG], then STK, which is the first menu key). Don't forget that menus are shown in pages of six functions each. Other pages can be accessed by pressing [NXT] (next page) or [←][PREV] (previous page). The commands in this menu are as follows:

- OVER places a copy of the object found in level 2 on the stack:

```
4:
3:
2:                123
1:                456
```

  After pressing OVER:

```
4:
3:                123
2:                456
1:                123
```

- ROT rotates the 3 first stack entries:

```
4:
3:                  3
2:                  2
1:                  1
```

  After pressing ROT:

```
4:
3:                  2
2:                  1
1:                  3
```

- ROLL is a similar function, but it takes one argument (from level 1 of the stack) which is the number of objects to "roll."

  Thus, 2 ROLL is the same as SWAP, and 3 ROLL is the same as ROT.

- ROLLD is similar to ROLL except that it rotates the objects in the opposite direction. For example if the stack contained the following:

```
4:                 4
3:                 5
2:                 6
1:                 3
```

After pressing ROLLD:

```
4:
3:                 6
2:                 4
1:                 5
```

(Don't forget that ROLLD takes one argument, the 3).

- PICK also takes one argument from the stack. PICK copies the object found at that level and places it in level 1. So, 2 PICK would be the same as OVER. For example:

```
4:         123456789
3:                 1
2:                 1
1:                 3
```

After pressing PICK:

```
4:         123456789
3:                 1
2:                 1
1:         123456789
```

(remember that PICK takes one argument from the stack).

- DEPTH tells us the number of objects that are on the stack. If the stack were empty, DEPTH would return 0. For example:

```
4:
3:
2:             33333
1:             44444
```

After pressing **DEPTH**:

```
4:
3:                33333
2:                44444
1:                    2
```

(there were 2 objects on the stack).

- DUP duplicates the object found in level 1:

```
4:
3:
2:                    2
1:                    1
```

After pressing **DUP**:

```
4:
3:                    2
2:                    1
1:                    1
```

- DUP2 duplicates the first 2 objects of the stack:

```
4:
3:
2:                    2
1:                    1
```

After pressing **DUP2**:

```
4:                    2
3:                    1
2:                    2
1:                    1
```

- DUPN is a generalization of DUP and DUP2. It takes an argument (N) and duplicates the first N objects of the stack.

  Thus, 1 DUPN is the same as DUP, and 2 DUPN is the same as DUP2.

- DROP2 "drops" the first 2 objects from the stack:

```
4:
3:                    3
2:                    2
1:                    1
```

After pressing **DROP2**:

```
4:
3:
2:
1:                    3
```

- DROPN is a generalization of DROP and DROP2. It takes an argument (N) and drops the first N objects from the stack.

  Thus, 1 DROPN is the same as DROP, and 2 DROPN is the same as DROP2.

Exercises

2-1. Calculate $\dfrac{5}{(3+1)\cdot(9-5)}$

2-2. If the stack contains:

```
4:
3:                    3
2:                    2
1:                    1
```

how would you arrive at the following stack?

```
4:
3:                    1
2:                    2
1:                    3
```

2-3. What would the following sequence of keys calculate?

[5] [ENTER] [3] [×] [1] [1] [−] [4] [÷] [1] [−] [COS]

What is the result?

3. Organizing Your Data Properly

The HP 48 is a true computer, and as such it must be capable of storing data—usually referred to as objects. These objects can be of different types: real numbers, binary integers, programs, lists, etc. They can be grouped into two families: internal objects (pre-programmed functions) and user objects (those that you enter into the machine). All objects will appear either on the stack or in the form of directory labels.

Menus and Directories

A menu or directory consists of a series of objects. Each object is accessible by invoking its name or by pressing one of the six keys at the top of the keyboard beneath the item in question.

For example, [←][MEMORY] (MEMORY) takes you to the MEMORY menu, which is a list of internal functions that provide memory management. Now, if you press [A] (the white button below MEM in the lower left corner of the screen), the machine returns a value on the stack. The screen should now look something like this:

```
4:
3:
2:
1:                 26173.5
```

When you pressed the [A] button, the HP 48 knew that you wanted to execute the object MEM, and it responded to your command. This function returns the amount of memory that is free for use, expressed in nibbles (see **Appendix B** for more about binary and hexadecimal notations).

If there are more than 6 objects in a menu, the others will appear by scrolling through the list using [NXT] (NEXT page) and [←][PREV] (PREVious page). Thus if you were to press [NXT], you would be able to use the other functions of the menu MEMORY (and if you continually press [NXT] in a menu, after you arrive at the last page of the menu, you are returned to the first page).

To give another example: [←][MODES] puts you in the MODES menu, which has 4 pages:

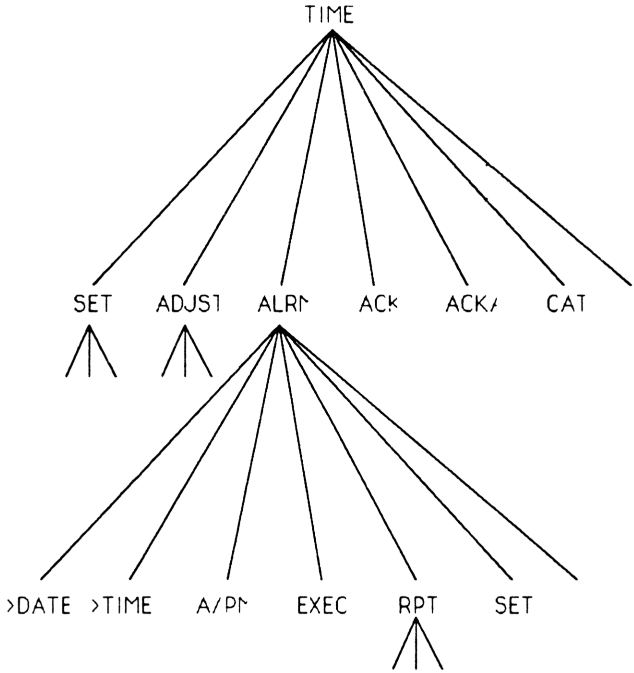
page 1:	STD	FIX	SCI	ENG	SYM	BEEP
page 2:	STK	ARG	CMD	CNC	ML	CLK
page 3:	DEG	RAD	GRAD	XYZ	R∠Z	R∠∠
page 4:	HEX	DEC	OCT	BIN	FM,	

If you press CLK (found at the end of page 2), the time and date appear (or disappear) at the top of the screen, and the label CLK becomes CLK■. When a "■" appears in a menu label, it means that the option in question has been activated. These menus allow us to personalize the HP 48 to function according to our own needs.

As mentioned in **Chapter 1**, certain menu labels will look like little folders. Such is the case for the PROGRAMS menu (accessed by pressing [PRG]). This means that if you press the corresponding button, you will enter a sub-menu of the current menu.

Menu Trees

The best way to explain a menu structure is by using the analogy of a tree. The first menu is called the root. In the root menu we will see "normal" labels and perhaps the special "folder" labels. These "folder" labels are parent menus that give us access to sub-menus. For example, the menu TIME (⬅ TIME), has this tree structure (partially represented here):



Sub-menus can contain objects, or they can have their own sub-menus (for example RPT is a sub-menu of the sub-menu ALRM), and so on. To distinguish the menus from one another, we refer to them as parent-menus and child-menus. These menus are connected by a branch; the parent being the one closest to the root, and the child is the one farther from the root.

The VAR Menu

There are two types of menus: menus of built-in objects and user menus—where you can store objects of your own choosing. The "VAR" menu is your user menu. Here is where you may store your own objects, create your own directories, etc. The root directory of the VAR menu has a special name: HOME.

To enter a subdirectory, simply press the key that corresponds to the subdirectory label (a "folder" label)—or, alternatively, type the name in full. To return to the parent directory, press [←][UP] (UPDIR); to return directly to the root, press [→][HOME] (HOME). The directory that you are in at any instant is referred to as the current directory.

To store an object, simply place it on the stack, and enter a name by typing the letters between single quotes, and press [STO] (STOre). For example, press [5][1][2][ENTER], which places the real number 512 on the stack. Then press ['][α][α][A][B][C][ENTER]. The screen should show:

```
4:
3:
2:                  512
1:                'ABC'
```

Now press [VAR], to place you in your working directory, then [STO], to store the number. [ABC] should appear to the left of the current directory menu.

To recall this object, simply type ['][α][α][A][B][C][ENTER] [→][RCL]. You may also type [→][ABC] or simply [ABC] (that is, press the white menu button below the [ABC] label). Thus, to recall the real number 512 previously stored, press the menu button for [ABC].

If the name ABC already exists in the directory, you can store something different under that name (which will erase the previous contents). To do so, you simply place the new object on the stack and press [←][ABC].

To create a subdirectory, use the **CRDIR** command, found in the MEMORY menu. You type the name of the intended new directory (for example 'DIREC'), then press **CRDIR**.

By creating subdirectories, you can group related objects together in one area. For example, if you have stored mathematical programs, machine language programs, and games, it would be wise to create 3 subdirectories in the HOME directory: **MATH**, **ML**, and **GAME**. This allows you to find each of your programs easily and quickly.

Three additional commands are important to know when working with directories:

UPDIR ([←][UP]) lets you go "up" to the parent of the current directory

HOME ([→][HOME]) lets you go directly to the HOME directory (the root directory of VAR)

PATH (in the MEMORY menu: [←][VAR] **PATH**) permits you to see where you currently are in the VAR tree structure. This command returns a list containing the names of directories (the first of which is always HOME).

Exercises

3-1. Create a subdirectory **EXD** in the HOME directory and place in it three variables **A**, **B** and **C**, containing the real numbers 1, 2 and 3, respectively.

3-2. How many sub-menus are in the MTH menu?

4. Programming the HP 48

Besides using the many internal functions of the HP 48, you can also create your own functions from them. The HP 48 has a true programming language, called RPL (Reverse Polish Lisp), derived from the language, LISP ("LISt Processor"—a.k.a. "Lots of Insane and Stupid Parenthesis"). LISP is very powerful (used for artificial intelligence), but its syntax is very difficult, because every command is coded between parentheses. The vast amount of parentheses in its programs make it very difficult to read.

However, Reverse Polish Notation, as you have seen, allows us to work without parentheses—by using **objects**. That term is intentionally vague: The HP 48 makes the least possible distinction between the *types* of the objects that it manipulates. The functions adapt to their given inputs. For example, if the stack contains the real numbers 2 and 3...

```
4:
3:
2:                     2
1:                     3
```

...pressing ⊞ gives the proper result of 2+3:

```
4:
3:
2:
1:                     5
```

But if you place "ABC" and "DEF" on the stack...

```
4:
3:
2:                 "ABC"
1:                 "DEF"
```

...then ⊞ will "add" (i.e. concatenate) the two strings, giving this result:

```
4:
3:
2:
1:              "ABCDEF"
```

Thus the same + operation will add two real numbers, two binary integers, two matrices, or a real and a binary, a character string and a list, etc. This generic adaptation of functions makes complicated programming easier .

Programming Methods

As we have seen, a program is a group of commands. In the case of RPL, this group of commands is given between two symbols: « and ».

For example, to calculate the cube of a number, we would enter the number, then this sequence: [3][y^x]. But to calculate many such cubes, it would be nice to simplify this procedure—create the program CUBE1.

- To begin the program, we must enter a special character, «, by pressing [⬅][« »]. As you can see, the closing delimiter (») is also present. The screen should now look like this*:

```
2:
1:
«
»
```

  There is also a blinking cursor to the right of the «. It is here that the next characters will be entered.

- The program's first step is to place a 3 onto the stack, so press [3], and a space ([SPC]) which will serve as a separator.
- The second command is y^x, so press the [y^x] button. You may expect y^x to appear, but instead you see the symbol ^. This signifies "raise to the power." The screen should now show this:

```
2:
1:
« 3 ^
»
```

  And the cursor should be to the right of the ^.

*Note: If you make a mistake while entering the program, the [◆] button allows you to erase the character to the left of the cursor. In the case of a more devastating mistake, pressing [ATTN] (that's the [ON] key) will erase everything you have entered—without destroying the contents of the stack.

- Our program is finished, so enter it onto the stack by pressing [ENTER]. The screen should now show:

```
4:
3:
2:
1:                « 3 ^ »
```

The program is now on the stack, and it is in level one. W e could now execute the program by pressing [EVAL], but this would cause an error (since the stack doesn't contain enough arguments), and we would lose the program (once executed, it disappears from the stack).

So before trying to use it, we will store it in a variable by entering the following sequence: ['][α][α][C][U][B][E][1][ENTER][STO]. Now, if you press the button [VAR], you will see **CUBE1** in a label in the left of the menu. This is your program.

Now enter a number onto the stack, press the button directly below **CUBE1**. The number on the stack will be cubed—with the touch of cne button instead of three!

There are other ways to program such a procedure. Here are a few examples—presented as are the programs in the library (**Part Three**):

CUBE2 (# D649h)
```
«
   DUP DUP * *
»
```

CUBE3 (# E4F0h)
```
«
   → A
   «
      A A * A *
   »
»
```

CUBE4 (# 4526h)
```
«
   → A
   'A*A*A'
»
```

This listing is interpreted in the following manner:

- The name of the object (or program) is in bold letters;
- After the name, in parentheses, is the object' s ***checksum*** value, to help verify that the object was entered correctly. To calculate the checksum, place the name of the object on the stack (e.g. 'CUBE2') and execute BYTES. This function returns two values: the checksum and the object's size. (The checksums here are in hexadecimal, so to make comparisons, put your HP 48 in this mode by typing HEX.)
- Below the object name is the listing, as it would appear after entry .

To enter these objects, you must:

- Type the object (just as with CUBE1) and enter it onto the stack;
- Enter its name onto the stack;
- Press STO.

A few notes on these four programs:

- CUBE1 uses the pre-programmed internal function, the power notation ^, which takes two arguments from the stack: a real number and the power to which you would like to raise it. CUBE1 places the power onto the stack (in this case 3); it's up to you to supply the real number.
- CUBE2 uses the stack. The DUP function duplicates level 1 of the stack. (It is very rapid, as are all stack functions.) Executing DUP twice gives 3 copies of the object, which are then multiplied together. For example, if CUBE2 were executed with this stack:

```
4:
3:
2:
1:                    5
```

After the first DUP we would have:

```
4:
3:
2:                    5
1:                    5
```

...after the second DUP:

```
4:
3:                    5
2:                    5
1:                    5
```

...after the first multiplication:

```
4:
3:
2:                    5
1:                   25
```

...and after the second multiplication, the cube of 5:

```
4:
3:
2:
1:                  125
```

- CUBE3 uses the "local variable" concept. We have already seen variables stored as objects in the VAR menu. A local variable is visible only to the program in which it is declared. To create such a variable, we use the symbol →, followed by one or more variable names, then a « to signify the end of the list of names. This will create local variables—using the values that were on the stack—from that point on in the program until a matching » delimiter is reached. In that part of the program, any use of a name of one of these variables will recall the value given by →. A few notes on local variables:
  - → conserves the order that the numbers were placed on the stack. If the stack has a 5 in level 2 and a 42 in level 1, then → A B will place 5 in the variable A and 42 in the variable B.
  - If a local variable has the same name as another variable, the contents of the most local variable are used. For example, in the following program: « 1 → A « 2 → A « A » » »
    1 is placed in the first local variable A, then 2 in a local variable of the same name. When A is recalled, its value is 2.
  - All local variables will disappear when the program terminates, whether the program terminates normally or by interruption.
  - While local variables are visible only locally, global variables appear in the VAR menu and can be used from anywhere.
- CUBE4 is similar to CUBE3, but instead of a program object, the → A is followed by an algebraic that accomplishes the same task.
- CUBE1 is the shortest of the four, but if the user forgets to give an argument on the stack, he will get this error message: ^ Error: Too Few Arguments. Also a 3 will be left on the stack, and this is not very "clean." By contrast, the other programs begin with a function that first tests for the presence of an object on the stack.

  The following program is the shortest, gives the best performance, and is the most correctly programmed. :

  CUBE (# C875h)

```
«
   → A
   'A^3'
»
```

- As a general programming rule, you will need to choose between the methods in CUBE2 and CUBE3, knowing that CUBE3 is programmed well because of its use of local variables to store arguments, and its use of the stack for calculations; but it is slower than CUBE2 because recalling a local variable is slower than executing a DUP.
- You must avoid, at all costs, this method of programming:

  ```
  « 'A' STO A A * A * 'A' PURGE »
  ```

  This is very slow because it creates and purges a global variable, and it may erase a preexisting global variable, A. Even so, such a method is occasionally necessary.

Variables and Directory Trees

We have seen that a local variable is visible only in a certain section of a program, appearing at the beginning of execution of this section and disappearing at the end. We have seen that a global variable is an object stored in the VAR menu or in one of its subdirectories.

Variables can have identical names. You can have global variables of the same name (in different directories as well as local variables with that name). Which value will be used when we recall a variable? To understand this, we must understand how the HP 48 searches its contents:

- First step: The HP 48 checks for any local variables of the specified name, beginning with the local variables most recently created.
- If a local variable is not found, it looks for the name in the current directory. If it finds it, it's done. If not, then if the user is not in the HOME directory, the HP 48 checks the parent directory. If it gets to HOME without finding the variable, then instead of using the contents of the variable, it uses the name (between single quotes ' '). (For a more detailed discussion of directory trees, see **Chapter 3**).

The HP 48's capacity to manage local variables permits a classic programming technique: recursion.

Recursion

Certain mathematical problems use recursion. That is, they refer to themselves. For example, the calculation of a function f on a point n could be:

- $f(n)=g(f(n\text{-}1))$, where g is a known, calculable function.
- $f(n_0)=f_0$, a known value.

We are perfectly capable of calculating $f(n)$, for any n greater than n_0. We simply apply the first formula repeatedly. If $f(n_0)=f_0$ is known, then so is $f(f(n_0))$, and $f(f(f(n_0)))$, etc. In other words, to calculate $f(n)$, we use $f(n\text{-}1)$ to make the calculation; to calculate $f(n\text{-}1)$, we use $f(n\text{-}2)$, and so on.

Let's calculate, for example, the factorial function:

- factorial(n) = $n \times$ factorial(n-1);
- factorial(0) = 1.

That is, to calculate factorial(n), we say:

- "If $n = 0$, we know this, it is 1."
- "If $n > 0$, we must calculate factorial(n-1) and multiply this by n."

This can be programmed directly:

FACTORIAL (# 3386h)

```
« → N
  « IF
      N 0 ==
    THEN
      1
    ELSE
      N 1 - FACTORIAL N *
    END
  »
»
```

First we take a value from the stack and place it in the variable N. Next we test if N is equal to 0. If so, we know the solution and return the value 1 to the stack. If not, we calculate factorial(N-1) and multiply it by N.

To better understand the operation of a recursive program, you must understand that when a program "calls itself," it executes a *copy* of itself—a copy that has nothing to do with the original. Look, for example, at the calculation of factorial(2). To calculate this we will need the values of factorial(1) and factorial(0)—which we already know. Thus, 3 copies of FACTORIAL are chained together. Observe:

Copy 1	Copy 2	Copy 3
This is the copy we call with the value 2 on the stack. In this case, N has the real value of 2. N ≠ 0, so to find factorial(N-1), it puts the value (N-1=1) on the stack and calls factorial.		
It now waits for a response.... N is still 2.	Factorial begins with a 1 as the N value for the function. Again, N ≠ 0, so it finds factorial(N-1) by putting that value (N-1=0) onto the stack and again calling factorial.	
Still waiting; N is still 2.	Waiting here, too; N is still 1.	Factorial begins with N = 0. But factorial(0)=1, so the value of 1 is returned immediately to the calling program.
Still waiting; N is still 2.	The value of factorial(0) arrives and is multiplied by N to get 1.	
Finally, the value of factorial(1) arrives and is multiplied by N to get 2.		

The principle is the same regardless of the value of the first N. Look at this summarized example for 5. In all, there are six copies of the factorial program in action:

Copy 1	Copy 2	Copy 3	Copy 4	Copy 5	Copy 6
N=5, *f*(4)=?					
N=5...(wait)	N=4, *f*(3)=?				
N=5...(wait)	N=4...(wait)	N=3, *f*(2)=?			
N=5...(wait)	N=4...(wait)	N=3...(wait)	N=2, *f*(1)=?		
N=5...(wait)	N=4...(wait)	N=3...(wait)	N=2...(wait)	N=1, *f*(0)=?	
N=5...(wait)	N=4...(wait)	N=3...(wait)	N=2...(wait)	N=1...(wait)	N=0, *f*(0)=1
N=5...(wait)	N=4...(wait)	N=3...(wait)	N=2...(wait)	N=1, *f*(0)=1 -->*f*(1)=1	
N=5...(wait)	N=4...(wait)	N=3...(wait)	N=2, *f*(1)=1 -->*f*(2)=2		
N=5...(wait)	N=4...(wait)	N=3, *f*(2)=2 -->*f*(3)=6			
N=5...(wait)	N=4, *f*(3)=6 -->*f*(4)=24				
N=5, *f*(4)=24 -->*f*(5)=120					

Thus we find that factorial(5)=120.

Exercises

4-1. Write a program that will add two real numbers taken from the stack. Would it also work for two strings?

4-2. What does the following program do?

```
« → A B « A B + A B * / » »
```

4-3. Write a recursive program to calculate the n^{th} term of the Fibonacci series U_n defined by:

- If n is greater than or equal to 2, $U_n = U_{n-1} + U_{n-2}$;
- $U_0 = U_1 = 1$.

5. Presenting Your Data Properly

So far, we have discussed the calculation capabilities, data storage, and programming of the HP 48. But simply knowing these is not sufficient.

The memory of the HP 48 can be quite large. It has 32 Kb of base RAM, which can expand up to 288 Kb with two 128 Kb cards—the equivalent of more than 200 pages of text. Therefore, it is important to be well organized and to present your programs and data in a manner that will make it easy to find them later. To do this, there are a few techniques that we will now study.

Making Data Access Easier

In **Chapter 3**, we studied menu and directory tree structures. This is an essential element of organizing programs and data, because the tree structure allows you to group similar classes of variables and programs together. For example, Mathematical programs together in a 'MATH' directory, matrix programs in a subdirectory, etc.

In any subdirectory, it is possible to order the variables and programs with the function ORDER. This command takes, as its argument, a list containing the names of the variables in the desired order. The function then puts them in that order. In this way, for example, you can place the important programs first, followed by sub-programs that are less useful.

It is also essential to choose program names carefully, so that simply seeing the title of a variable or program will suggest its contents. Occasionally, however, it is useful to associate a name of a pre-existing function or an icon to a program that we have just written. This is made possible by using a CuSTom menu (via the CST button—next to VAR).

A custom menu permits us to connect objects of the HP 48 and a specific menu label, without excessive memory consumption. The mechanism behind this menu is simple: when you press the CST button, the HP 48 searches for a variable named CST.

If the variable is not found in the current directory, the HP 48 searches the parent directory(s) until it reaches the root. If no variable CST is found, an empty menu is shown. Therefore, it is possible to have many dif ferent CST menus, depending on which directory you are currently in (which reinforces the notion of good data organization).

The variable CST must contain a list. For each element of this list, we have many possibilities:

- A name: The menu label is associated with the variable of that name.
- A string of characters: The string is placed in the command line when that menu key is pressed.
- A list of two objects: The first object is the title of the menu label; the second is the associated object. If the first element is a 21 ×8 graphics object, the menu title is the corresponding graphic.
- All other objects will be executed. The object will appear in the menu label for the corresponding button.

Here is an example of a CST menu:

```
CST (# 9D17h)
  { { "A" "Un " } { GROB 21 8
  0000000400C10A00E08FFFF0EFFFF1F700C10CFF70000000
  "avion " } { "in" "dans"} { "the" "le "}
  {"sky" "ciel "} "!" }
```

After storing this object, enter the CST menu (by pressing CST, to the left of VAR). Interesting, no? Now press in succession the six menu keys from left to right. Your HP 48 has just accomplished an English-French translation!

A custom menu permits us to associate icons with functions. It also permits us to mix the HP 48 internal functions with user functions on one menu. But we can even do better than this. There is also a way to assign functions to any key on the keyboard.

This method of redefining keys is best described through an example. Here is a small program that plays an tune of random music:

```
«
   -56 CF 1 10
   START
     4400 RAND * .1 RAND * BEEP
   NEXT
»
```

Type that in. The screen should look like this:

```
2:
1:« -56 CF 1 10 START
   4400 RAND * .1 RAND
   * BEEP NEXT »
```

Now type: [5][1][ENTER][A][S][N][ENTER]. Then press [←][USR], then [ENTER], and you will hear a little music.

The explanation for this is simple. We have assigned this particular program to the [ENTER] key. This assignment is not valid except in USER mode. We entered this mode temporarily by pressing [←][α] (this sequence puts us in 1USR mode, that is, USER mode for *one keypress*). To remain in USER mode, type [←][USR][←][USR], and **USER** will appear at the top of the screen. To return to normal mode, type [←][USR] once again.

Note: Any keys that are not defined for USER mode retain their original functions in USER mode.

You may redefine the entire keyboard, including the [ON] button. The syntax for ASN is the following:

arg1 arg2 ASN

arg1 is the function that you would like the machine to execute when you press the key. This can be the name of a program, the program itself, or a completely different object.

arg2 is a real number composed as follows:

- The first digit (tens position) is the key's row (a value between 1 and 9, where 1 is the top row of keys);
- The second digit (ones position) is the key's column (a value between 1 and 6, where 1 is the left-most column of keys);
- The decimal place is the button mode:
  - 0 or 1 normal mode
  - 2 [←] mode (orange shift)
  - 3 [→] mode (blue shift)
  - 4 [α] mode (alpha)
  - 5 [α][←] mode (alpha, orange shift)
  - 6 [α][→] mode (alpha, blue shift)

For example, to redefine the [DROP] key, you would assign a new function to the button 56.2.

Note that to restore a key to its standard function, you use the special predefined name, 'SKEY'. Or, executing 0 DELKEYS will return *all* buttons to their standard functions.

Understanding Programs More Easily

Many methods exist to increase the understanding of programs or their results. We will mention three important and easy-to-use methods:

- The HP 48 allows you to enter comments that begin with the character @ ([α][↱][ENTER]). Unfortunately, these comments disappear as soon as you press [ENTER]. Therefore, they are not very useful unless you are storing the programs on another computer. To leave comments in a program more permanently, you can enter the following: "*comment*" DROP, where "*comment*" is the desired text. This type of comment will remain in the program. Thus you can note the purpose of the program, its syntax (e.g. the number of arguments it needs), and what results it will return.
- Messages: It is good to tell the user what is going on once in a while. For example, you can include error messages or indicate how (or what) the program is doing in the case of lengthy calculations.
- Explain the results: What is more frustrating than a program that returns data of whose meaning we have no idea? To easily remedy this, it is useful to "tag" the results—add a prefix to them (name, comment, etc.) via a special HP 48 function: The function →TAG takes as its arguments the object to be tagged, and its tag. The program mSOLVER in the library of programs uses this technique.

Above all, remember that you should always write your programs as if someone else must use them. In this way, if sometime later you decide to look at them again, you should not encounter too many dif ficulties.

6. Saving and Transmitting Data

The memory of the HP 48 is not infinite. The default amount is only 32 Kb (32 Kilobytes is about 32,000 characters). For this reason, it may be necessary to increase the memory by using RAM cards. In the HP 48SX, two ports are provided for this purpose (found on the back of the machine underneath the cover at the top).

But even if you don't need more memory, the HP 48 also allows you to easily load information from other machines. After all, why re-type data or programs that already exist on another HP 48? This is no fun, and errors are easily made in the process. It is much more useful to exchange data directly between machines or store the programs on a computer.

Plug-In Cards (HP 48SX)

There are two types of plug-in cards: ROM and RAM.

ROM is memory that you can only read (Read Only Memory). Its information cannot be modified. There are actually four types of ROM:

- real ROMs, (like those contained in the HP 48);
- PROMs or Programmable ROMs;
- EPROMs which are PROMs that can be erased by ultra-violet light;
- EEPROMs which are Electronically Erasable PROMs.

The EEPROM type of card is the most common, and it is sold pre-programmed (e.g. the HP SOLVER card). You could actually make one of these yourself (using an EPROM or EEPROM), but it would be costly.

RAM is memory that you can modify (Random Access Memory). Existing plug-in RAM cards for the HP 48 are 32 Kb or 128 Kb. On each of these cards is a small switch that allows you to write-protect it (like transforming it into ROM). These cards can be useful in two different ways:

- They can be used as a memory extension using the internal function MERGE. To put a card in MERGE mode, turn the machine off, insert the card in one of the two ports of your choice and turn the machine on. Then type 1 MERGE or 2 MERGE, depending on whether you placed the card in port 1 (the one on the bottom with the calculator upside down) or in port 2. At this point, type MEM, and if all is well, your memory will have been increased considerably.
- They can be used as a RAM disk in BACKUP mode. To put a card in BACKUP mode, insert the card in a port, and store your data directly on the card. The names of the objects of a port are not of the form '*name*' but are "tagged" objects in this form: :*x*:*name* where *x* is the number of the port (0, 1 or 2). For example, if the card is in port 2, then "hello":2:BONJOUR STO will store the string "hello" under the name BONJOUR in port 2. When storing, the card must not be write protected. It is wise to leave a backup card write protected unless you are in the process of storing data.

We must mention three important notes:

- A card in MERGE mode must not be write protected.
- A card in BACKUP mode that is write-protected <u>is not affected by a 'memory lost'</u>.
- If a card is installed in one of the ports, it is not "merged," and no data has yet been stored on it, you will get the message Invalid Card Data when you turn on the machine. This is because the card has not yet been configured.

HP 48 <-> Computer: RS-232C

HP sells a cable that connects your HP 48 to a Macintosh, an IBM-compatible computer, or any computer with a standard (9 or 25 pin) RS-232 serial port. Software is included with the cable to let you to save the data of your HP 48 on a hard or floppy disk. This software is called KERMIT.

You may transfer data in either direction:

- Transferring data from the HP 48 to the computer:
  - on the HP 48: '*name_of_the_object_to_send*' SEND
  - on the computer: **RECEIVE**.
- Transferring data from the computer to the HP 48:
  - on the HP 48: RECEIVE (I/O menu)
  - on the computer: **SEND** *name_of_file_to_send*

For any transfer, you should always make sure that the I/O parameters are set to what you really need. Here is a good configuration:

- On the HP 48, enter the I/O menu and press SETUP, then, by pressing the proper buttons, make your screen look like this:

```
    I/O setup menu
IR/wire:             wire
ASCII/binary:        ascii
baud:                9600
parity:              none 0
checksum type:       3
translate code:      1
```

- On the computer, you must be certain that the corresponding settings are the same as above. In particular, on IBM PC compatibles, you may type the following commands (after running Kermit each time, and before the first transmission):

SET BAUD 9600
SET PORT 1

Infrared Transfers

Two HP 48 machines may exchange data without any wire connections if they are less than 2 inches apart. To do this, the two machines must have the same SETUP.

For example:

```
    I/O setup menu
IR/wire:            IR
ASCII/binary:       ascii
baud:               9600
parity:             none 0
checksum type:      3
translate code:     1
```

In particular, note that the transfer mode must be IR (Infra Red) instead of wire, as with the connection to a computer.

Place the two machines head-to-head with the two little arrows pointing to each other (the arrows are found just above the second 'T' in "HEWLETT-PACKARD"). At the same time, enter '*name_of_the_object_to_send*' SEND on the sending machine, and RECEIVE on the other.

The object sent will be stored in the current directory of the receiving machine. If that name already exists in the current directory of the receiving machine, the object will be stored with a new name in the form *original_name*.1 (then *original_name*.2 and so on with each transfer of an object with the same name), unless flag -36 is set. Type -36 SF to set the flag, and -36 CF to clear the flag. If the flag is set, then the old object will be erased by the new one.

Caution: If the batteries are low, then transfers will not work properly.

Notes

7. Other Strong Points of the HP 48

The HP 48 is above all a scientific calculator and we will see some of its capabilities as such in this chapter. This chapter is not to give an in-depth explanation of these functions, but rather to make you aware of their existence. In this way, if you desire further understanding, you may look these functions up in the manuals that were furnished with the machine.

Symbolic Calculations

The HP 48 is capable of "symbolic" calculations. That is, the HP 48 is not limited to numeric calculations only, but is capable of applying complex mathematical operations directly to literal expressions. Some examples:

- Derivatives: To take the derivative of an expression with respect to a variable, type: '*expression*' '*variable*' [→][∂]

  Thus, 'SIN(X)/X' 'X' [→][∂] returns 'COS(X)/X-SIN(X)/X^2'

  Caution: If a value is stored in a variable 'X' of the current directory or one of its parent directories, the expression will be *evaluated*; you will not obtain the desired *symbolic* result. In this case, you must purge the variable 'X' or use a different variable in the expression.

- Taylor's Approximation: '*expression*' '*var*' *n* TAYLR
  where '*expression*' is the algebraic expression you want to integrate, '*var*' is the dependent variable, and *n* is the order of the polynomial with which the approximation will be made.

  Example: 'SIN(X)' 'X' 5 TAYLR returns:
  'X-1/3!*X^3+1/5!*X^5'

  Note: TAYLR is found in the ALGEBRA menu ([←][ALGEBRA]).

- Solving equations; finding extrema; calculating the value of a function on a point; all these may be done with the functions found in the SOLVE menu ([←][SOLVE]).

Numerical Calculations

The HP 48 possesses many functions useful in numerical calculations (and the list is too long to do justice here). Most of these functions are found in the MTH menu and are grouped into six categories: fraction calculations, probabilities, hyperbolic calculations, matrix calculations, vector calculations, and binary integer calculations (in dif ferent bases). There are also many statistical functions that are available in the ST AT menu (⬅ STAT).

The HP 48 uses 12 significant digits to give you a numeric result as accurate as possible. Internal calculations are done with as many as 15 significant digits.

Note also that if the returned result could be represented in a fractional form, the function →Q (⬅ →Q) can convert the real number to the closest fraction.

Graphs

The PLOT menu (⬅ PLOT) has all the necessary functions for plotting curves of all kinds (classic, conical, polar, parametric, etc.).

Note that you can view and edit the current graph by pressing ⬅ GRAPH. You can move the cursor using the four arrow keys, copy the coordinates of the cursor to the stack by pressing ENTER, and return to normal mode by pressing ON. The many functions (zoom, moving blocks, plotting or erasing points, lines, circles, marking points, etc.) are all available in this menu.

Units

The HP 48 can do calculations with units. To create a unit object, simply enter a real number, then the underscore character (_ , obtained by [→][_]), followed by the characters representing the desired unit.

For example, to create `1_m`, you would press [1][→][_][α][←][M].

Alternatively, you can place just the value on the stack, then go to the UNITS menu ([←][UNITS]), and choose the desired unit from one of the 16 possible categories (length, area, volume, time, speed, mass, force, energy, power, pressure, temperature, electricity, angles, light, radiation, and viscosity).

[→][UNITS] gives you another UNIT menu with various functions including the CONVERT function which allows conversion between dif ferent units.

Time

The TIME menu ([←][TIME]) gives you access to a series of functions for the clock. In particular, you can set alarms and perform certain calculations at specific times or on specific days. Note that [→][TIME] gives you direct access to the alarm catalog.

Conclusion

What we have learned here is only the beginning of the great possibilities of the HP 48. These are just the basics as well as a few tricks to give you a general idea of the capabilities of the machine.

Use your machine as often as possible and study the HP 48 manuals to gain a better understanding of what has been covered in this "first approach." The more you practice, the easier it will become, and you will soon learn to rapidly resolve long and tedious problems.

When you become familiar with the uses of the HP 48 (as defined by Hewlett-Packard) you will realize that it is indeed a marvelous tool. But remember that this is not all there is to it! In **Part Two** you will discover that you can do much better using machine language programming!

Part Two:

Machine Language

Introduction

In **Part Two** we will not only learn how to write machine language programs, we will also learn how the HP 48 memory is organized. Every programmer who really wishes to use his machine to its fullest potential must have an excellent knowledge of its structure. This knowledge makes it possible to gain access to information needed—information that the designers did not necessarily intend to be accessed.

This guided exploration of the HP 48 will be done in several steps, including the lowest level, which is machine language. Machine language is the only language that the HP 48's processor can really understand and execute. We will also be studying the HP 48 on a higher level (the memory organization), with mention made of many objects used by the HP 48.

Basically we will learn:

- Machine language:
  - What is machine language?
  - The actual machine language used by the HP 48's Saturn microprocessor;
  - Machine language instructions (grouped by function type).
- The HP 48's objects:
  - Regular objects to which the user has access;
  - Internal objects undocumented by Hewlett-Packard.
- The HP 48's memory organization:
  - Memory in general;
  - The I/O RAM, or how to directly access the contrast, clock, screen, etc.;
  - reserved RAM that contains the HP 48's internal information;
  - User memory that contains the objects created by the user (programs, variables, etc.).
- How to program in machine language.

Some of these chapters will contain tables describing the calculator 's memory. In order to remain consistent, they will look like the following table:

address$_1$	*contents$_1$*	*length$_1$*
address$_2$	*contents$_2$*	*length$_2$*
address$_3$	*contents$_3$*	*length$_3$*
address$_{last}$		

What you should know:

- An address is a hexadecimal number (base 16) which is the position in memory of the contents contained in the table boxes. These addresses will always be organized in this manner: $(address_1) < (address_2) < (address_3)$. The table is read from top to bottom. If the object listed is not at a fixed address, the symbol @ will be used (often indexed with the form $@_i$ if more than one address is used) to indicate the starting address of the object. The last address ($address_{end}$) indicates the address of the first nibble following the last content entry of the table.
- The central column gives a brief description of what is contained in the specified memory area. The contents of this field are explained in more detail in the text accompanying each table.
- The length field (right column) indicates, in decimal, the number of nibbles of the table entry (note that a nibble is the basic memory element of the HP 48). Thus, $length_1 = address_2 - address_1$. This field may correspond to a specific value in one of the object fields. For example $length_1$ can be $contents_2$.

The first chapter of **Part Two** (**Chapter 8**) covers a general approach to machine language. If you are somewhat familiar with machine language, you will probably want to skip to **Chapter 9**.

Do not be overwhelmed by the vast amount of information found in **Part Two**, as it is mainly a reference guide. To best understand this material, the reading should be done twice. The first reading should be done rapidly to give you a basic understanding of the different ideas discussed. The second time should be taken more slowly, and you should try some machine language programming on your own as you go. You will then find that **Part Two** will be an excellent reference for future machine language programming.

8. Machine Language

If you are already familiar with what an assembler is and does, and you basically know what machine language is, then you may skip to the following chapter. Otherwise, you will find this chapter useful.

To explain the concept of machine language, we will compare it to a higher level language. Consider an analogy: a little story about Mr. Jones and Mr. Smith—two people each wish to install electrical outlets in their homes.

Mr. Smith is not a handy man, so the most simple solution for him is to call someone who is. He picks up the telephone and calls an electrician in his neighborhood. Later that afternoon, the electrician finally shows up at Mr. Smith's house and does the work for him for a considerable sum of money (materials + labor + travel + tips...). Mr. Smith pays grudgingly because the work was not done exactly as he would have liked.

Mr. Jones, on the other hand, is quite good with his hands, and he decides to do the work himself. He makes a trip to the hardware store where he buys a plug and some wire. Then, at home, he installs the plug how and where he wants it, all for a very modest sum of money.

You could say that in the first case, Mr. Smith used a high-level language by giving an order that resulted in a number of elementary operations being carried out (getting wire, getting a plug, installing, etc.). Mr. Jones, on the other hand, carried out these elementary tasks himself. He used a low-level language that was directly executable. It closely resembles machine language.

The story illustrates these two types of languages in these other respects, too:

- Calling the electrician is easier than doing the work yourself because you have only to give the orders!
- A high-level language is more costly in time (just as the electrician costs more money).
- Often a high-level language seldom does not let you do exactly what you want; you cannot ask for just anything (just as an electrician will probably not come to change a light bulb for you).

Machine language gives you direct access to all the available resources of the machine in an extremely fast but complicated way. It can do this because it is composed of very basic instructions. It is therefore necessary to use many instructions to carry out even the simplest functions.

Machine language is the only language that the machine really understands (thus all high-level languages are broken down into calls to programs written in machine language). However, if a language is easily understood by the machine, it is absolutely unreadable for a human being because it is composed of a series of numbers.

This is why we will introduce a third language: *assembly*. This language consists of a symbolic representation of machine language codes using mnemonics—abridged names that help you remember what function is executed by the machine instruction (for example, `P=0` instead of `20`).

But since the machine cannot understand these symbols, it is necessary to transform them into a series of numbers that are understandable. This translation of assembly to machine language is called assembling. The inverse operation is called disassembling. Thus we would begin by writing a program in assembly, then we would assemble it to make it executable by the machine.

For the HP 48, we can do the assembling by hand, or automatically using a more powerful computer. (There are at least two Saturn assemblers: Areuh for the IBM PC and UNIX machines, written by Pierre David and Janick Taillandier; and Satas for the Atari St, Amiga, IBM PC and UNIX machines, written by Christophe Dupont de Dinechin). A disassembler that works on all HP 48 calculators is given in the library of programs.

The last term to define is the "microprocessor." This is basically the heart of the machine, the electronic entity that executes the machine language instructions.

The basic unit of information recognized by the microprocessor is the bit (which can only be a value of 0 or 1). Because the machine uses a binary base, it is best for us to use a base that is a power of 2, which is why base 16 (hexadecimal) is used. The digits of base 16 are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, etc. Therefore, the value 23h (the 'h' signifies that the number is in hexadecimal) is equal to 35 in decimal (16 * 2 + 3).

However, it may sometimes be necessary to store numbers in decimal. We can use a notation called "binary coded decimal." This notation uses a hexadecimal number as if it were decimal. For example, the number 15h would be equal to 15 decimal.

This type of storage makes it necessary to have two dif ferent calculation modes for the microprocessor: hexadecimal mode, where the registers contain hexadecimal numbers, and decimal mode, where the registers contain "binary coded decimal" numbers.

The current mode determines the manner in which the mathematical operations are executed by the microprocessor. If you add the two numbers 9h and 3h in hexadecimal mode, the answer is Ch. If you add them in decimal mode, the answer is 12h, which corresponds to the decimal value 12 in "binary coded decimal" notation.

Exercises

8-1. Convert these decimal numbers into hexadecimal: 1, 10, 25, 65535, 48830.

8-2. Convert these hexadecimal numbers into decimal: 123h, 10h, 100h, B52h, 3h.

9. The Saturn Microprocessor

The HP 48 contains a 4-bit Saturn microprocessor. It is the same microprocessor as in the HP 71 and the HP 28.

The Registers

The Saturn microprocessor has 19 registers. A register is a memory location in the microprocessor and can contain only binary integers. These 19 registers can be grouped into six categories:

- I/O registers (2);
- Flag registers (3);
- Data pointer registers (3);
- Scratch registers (6);
- Working registers (4);
- Field pointer register (1).

The I/O Registers (2)

- **INPUT** (16 bits). This register is used to read the state of the 16 inputs (particularly useful for reading the keyboard).
- **OUTPUT** (12 bits). This register is used to send current to one or many of the 12 wires of the keyboard and the speaker. This register can only be written to.

These two "registers" are used for the BEEP sound (writing to **OUTPUT**), as well as for sampling the keyboard. To sample the keyboard, current is sent to a row of buttons. If current is detected in a column of buttons, this lets us know that the button at the intersection of the row/column is being pressed.

The table opposite shows each OUT/IN mask to test if a particular key is pressed (all the values are given in hexadecimal). To test a button, write the corresponding OUT, read the value coming IN, and AND this value with the value given in the table. If the result is non zero, this signifies that the button in question is pressed. It is possible to test many keys simultaneously by using an output mask constructed by ORing many masks together. (Caution: this method does not work for testing the ON button. Interrupts are needed for this, and we will study those later.)

Here are a few examples:

- To test if the button "A" has been pressed, send an OUT #002h, and read the value coming IN and do a logical AND with the mask #0010h. This is done with a small program:

  ```
  LCHEX  #002       output mask
  OUT=C
  GOSBVL #01160     this is C=IN
  LAHEX  #00010     input mask
  A=A&C  A
  ?A=0   A
  GOYES  Key_not_pressed...
  * key A is pressed
  ```

  Note: the routine at #01160h is used instead of the instruction C=IN because the latter does not function properly when used with RAM (it corrupts the memory area that was read). Another useful address is #01EECh, which successively executes OUT=C and C=IN.

- To test if any key has been pressed: The program above can still be used, but the output mask would become #1FFh (#001h OR #002h OR #004h OR #008h OR #010h OR #020h OR #040h OR #080h OR #100h); and the input mask #003Fh (#0001h OR #0002h OR #0004h OR #0008h OR #0010h OR #0020h).

- To emit a sound: alternate between output masks #800h and #000h (to activate and deactivate the speaker).

A	B	C	D	E	F
002 / 0010	100 / 0010	100 / 0008	100 / 0004	100 / 0002	100 / 0001
MTH	PRG	CST	VAR	▲	NXT
004 / 0010	080 / 0010	080 / 0008	080 / 0004	080 / 0002	080 / 0001
'	STO	EVAL	◀	▼	▶
001 / 0010	040 / 0010	040 / 0008	040 / 0004	040 / 0002	040 / 0001
SIN	COS	TAN	$\sqrt{x}$	y^x	$1/x$
008 / 0010	020 / 0010	020 / 0008	020 / 0004	020 / 0002	020 / 0001
ENTER		+/−	EEX	DEL	⬅
010 / 0010		010 / 0008	010 / 0004	010 / 0002	010 / 0001

α	7	8	9	÷
008 / 0020	008 / 0008	008 / 0004	008 / 0002	008 / 0001
↰	4	5	6	×
004 / 0020	004 / 0008	004 / 0004	004 / 0002	004 / 0001
↱	1	2	3	−
002 / 0020	002 / 0008	002 / 0004	002 / 0002	002 / 0001
ON	0	•	SPC	+
400 / 8000	001 / 0008	001 / 0004	001 / 0002	001 / 0001

OUTPUT / INPUT masks for the keyboard

Flag Registers (3)

- **CARRY** (1 bit). This is the carry bit; when an operation results in a carry, this flag is set.
- **HST** (hardware status) (4 bits). This is a register with 4 flags (**MP** module pulled, **SR** service request, **SB** sticky bit, **XM** external module missing).
- **STATUS** (16 bits). These flags are like those accessible by RPL instructions SF and CF (but they are not the same). Flags 12 to 15 are used by the HP 48, but flags 0 to 11 are available for use in programs. This register is represented by ST.

Data Pointer Registers (3)

These registers are used to point to a particular memory area. They each have a length of 20 bits. The HP 48 is therefore capable of addressing 2^{20} nibbles (512 Kbytes). The three registers are:

- **D0** and **D1** (20 bits each). These are used for reading and writing to memory;
- **PC** (program counter - 20 bits). This register contains the address of the instruction currently being executed.

Scratch Registers (6)

There are two types:

- **RSTK** (return stack) (8 levels of 20 bits each): This is a stack with 8 levels used for saving addresses. This stack behaves exactly like the HP 48 RPL stack with the difference that even if it's empty, it contains zeros. It serves as an information backup, particularly for saving the return address from a call to a subroutine.
- **R0, R1, R2, R3,** and **R4** (64 bits each): these are primarily used for backing up the working registers.

Working Registers (4)

The registers **A**, **B**, **C** and **D** (64 bits each) are used for miscellaneous calculations. **A** and **C** are dedicated specifically for reading and writing to memory (they are therefore used in conjunction with D0 and D1).

Field Pointer Register (1)

The working registers **A**, **B**, **C**, and **D** are very long (64 bits) and few in number. They are therefore divided into smaller pieces—"fields," which can be used independently, if they don't overlap. This permits simultaneous calculations using only a few registers. Here is a table of the fields:

register's nibble number

<table>
<tr><td>F</td><td>E</td><td>D</td><td>C</td><td>B</td><td>A</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">W</td></tr>
<tr><td>S</td><td colspan="12">M</td><td>XS</td><td colspan="2">B</td></tr>
<tr><td colspan="11"></td><td colspan="5">A</td></tr>
<tr><td colspan="13"></td><td colspan="3">X</td></tr>
</table>

Thus, field **M** represents nibbles E to 3, **A** the nibbles 4 to 0, and **W** is the entire register, etc. The names of these field pointer registers are the same as those used by the HP 71. Each letter stands for the following name:

- **A** - **Address**: Field **A** is 5 nibbles long (which is the length of an address) and was intended to contain addresses;
- **B** - **B**yte: Two nibbles equal one byte;
- **M** - **M**antissa: On the HP 71, a real number was stored in a register containing the sign, mantissa, exponent sign, and exponent. This is the mantissa field.
- **S** - **S**ign: Corresponds to the sign field of the HP 71;
- **X** - e**X**ponent: Corresponds to the exponent field of the HP 71;
- **XS** - e**X**ponent **S**ign: Corresponds to the HP 71 exponent sign field;
- **W** - **W**ide: In other words, the entire 64 bit register.

The length and position of those fields are fixed. However , there are two other fields, **P** and **WP** (for **Wide-P**). The size of **WP** depends on the contents of **P**. **P** is one nibble in length, and can therefore contain a number from 0 to F. **WP** will contain the nibbles 0 to P (see the table below). Note also that the register **P** also affects the way values are loaded into registers **A** and **C** (see instructions `LAHEX` and `LCHEX` in **Chapter 10**).

In an assembly program, the name of the intended field is written after an instruction. For example: `?C=0 A` means: "Is the field **A** of register **C** equal to zero?" There are two possible methods of indicating a specific field in an assembly instruction:

- The code for the operation actually exists and can be given directly . This is always the case for the **A** field, and sometimes for the **B** field.
- The code may be given as a small letter (a, f, or b) to be replaced by the code for the desired field according to the table below .

Example: If you have this line in the list of instructions: `Ab0 A=0 b` , for `A=0 W`, you would use the code `AF0` (F for W since the letter given is **b**).

Another way manipulate fields is to define the number of nibbles the operation will affect—indicated in the instruction list by an **x**. For example, `158x DAT0=A x+1` means that the operation will take place for *x+1* nibbles. Thus, `1583` would be "perform the operation `DAT0=A` for the nibbles 0...x of A). This type of operation is equivalent to using a WP field without having to change the value of the register **P**.

Field	**a**	**f**	**b**
P	0	0	8
WP	1	1	9
XS	2	2	A
X	3	3	B
S	4	4	C
M	5	5	D
B	6	6	E
W	7	7	F
A		F	

Miscellaneous Notes

The Saturn microprocessor has a peculiarity to be aware of: It reverses everything it reads. For example, if in memory location #00000h there is a 2, and in #00001h there is a 3, reading 2 nibbles from #00000h would return the value 32. For this reason, all values in memory must be written in reverse—for all reading and writing operations to and from the registers.

Saturn microprocessor instructions are listed using two dif ferent methods:

- By function type: This is useful when you are looking for a certain operation without knowing the exact syntax or the registers used. (This list is found in the following chapter).
- By code: This listing is found in the appendix, and is excellent as a reference card for programmers who are already familiar with how the operations work, or for someone who is disassembling an existing program.

One last note about the registers used by the HP 48:

- **D0** points to the next instruction to be executed (so we always finish a machine language program by writing to this address).
- **D1** points to the first level of the stack. Reading 5 nibbles from this address returns the address of the object in level 1.
- **B** points to the return stack. As we execute instructions, we may need to store return addresses. **B** points to the next free location in the return stack. (Caution: This stack is not the **RSTK** register).

These registers are used by the system. They may be used in a machine language program, but their original value must be restored at the end of program execution. The flags 12 to 15 are also used by the system (for interrupts), but, unlike the three system registers, they must never be modified. Note that Flag 15 is the one that can be used to change the way keyboard interrupts are handled. Flag 10 may be used and modified, but it is also used by the HP 48 for memory allocations. If we clear this flag before trying to reserve memory, it will be set if garbage collection was necessary.

Exercises

9-1. How would you code the W field for these instructions?

```
Ba3  D=D-C  a
AbB  C=D    b
```

9-2. How would you code the above using fields **P** and **WP**.

9-3. Knowing that:

```
Aa3  D=D+C  a
Ab3  D=0    b
```

disassemble the instructions A13, A73, A83 and A93.

9-4. If #00321h contains 1, #00322h contains 1, #00323h contains 4, #00324h contains C, and #00325h contains 8, what will your register contain after reading 3 nibbles from #00321h?

9-5. Given the same values as question **9-4**, what would your register contain after reading 2 nibbles from #00322h?

9-6. Given the same values as question **9-4**, what would your register contain after reading 4 nibbles from #00321h?

9-7. If field X of register A contains 210h (2 in nibble 0, 1 in nibble 1 and 0 in nibble 2) and you write this value to #70080h, what do memory locations #70080h, #70081h and #70082h contain?

9-8. If we then read 3 nibbles from #70080h into field X of register C, what will be the value contained in this field? Field **B**? Field **XS**?

9-9. If **P** equals 2, how many nibbles are implied by the instruction `A=DAT0 P` ?

10. The Saturn Instruction Set

This chapter covers the entire instruction set of the Saturn microprocessor. This list will allow you to easily find each instruction that you will need to write machine language programs. The instructions are grouped by functionality, as follows:

- Moves:
  - Immediate
  - Exchanging Register Fields
  - Saving and Restoring (**Rn** and **RSTK**)
  - Reading and Writing to Memory
  - Input and Output
- Exchanging Register Contents
- Arithmetic Operations:
  - Increment
  - Addition
  - Decrement
  - Subtraction
  - Logical AND
  - Logical OR
  - Logical NOT
  - 2's Complement
  - Multiplying by 2
  - Dividing by 2
  - Multiplying by 16
  - Dividing by 16
  - Rotating Left (one nibble)
  - Rotating Right (one nibble)
- Jumps:
  - Direct Relative Unconditional
  - Direct Relative Conditional
  - Absolute
  - Register Direct
  - Register Indirect
  - Getting the Program Counter

- Calling subroutines:
  - Direct Relative Unconditional
  - Absolute
  - Returning from Sub-routines
- Comparisons:
  - Immediate
  - Comparing Registers
- Bus Commands
- Control Instructions
- NOPs (Instructions with no effect)
- Pseudo Operations

Each operation is described as *instruction field (cycles) code* , where:

- *instruction* is the mnemonic for a particular instruction (e.g.: A=0);
- *field* is the field in which the instruction has effect;
- *code* is the hexadecimal code of the instruction.
- *cycles* is the number of CPU cycles needed to execute the instruction—very useful for calculating the exact time necessary to execute certain programs (tone generation, IR transmitting/receiving, etc.). Each CPU cycle lasts about 570 nanoseconds (the microprocessor speed is 1.7 MHz).

  The Saturn microprocessor is a 4 bit microprocessor, however the peripherals (ROM, RAM, screen controller, etc.) use 8 bits. For this reason there is a cache buffer between the microprocessor and the peripherals. This internal buffer is 2 nibbles long (one byte) at an even address location (for example, #00000h and #01234h are even address locations). The use of this cache buffer requires one clock cycle. The cache buffer is used when transferring machine language instructions from memory to the microprocessor. If the instruction is an odd number of nibbles, the number of memory accesses depends on whether the instruction is at an even or odd address location. For this reason, certain instructions will require n or $n+1$ cycles for execution. For this type of instruction, a speed of $n.5$ instruction cycles will be listed (4.5 for example). If the start of the address is even, then this value should be rounded down; otherwise it should be rounded up.

To make things even more complicated, instructions that read from memory also use the cache buffer. The number of cycles for such an instruction is listed in the form (n_1, n_2), where n_1+n_2 is the number of total cycles used for the instruction. The same rules apply for rounding n_1 as above, but if the number of nibbles read is odd, n_2 will be shown in fractional form. If the address of the area being read is even, then n_2 is rounded down; otherwise it should be rounded up. Certain instructions will have a different cycle time depending on how many nibbles they affect (field sizes are different, or reading and writing different nibble sizes to memory). For this case, q equals the number of nibbles the instruction affects. Finally, for comparison operations, two numbers are given in the form (n_1/n_2). The first is the number of cycles if the test is true, the second is if the test is false.

Example: Calculate the execution time of a loop. Here is a small assembly program:

```
L1   97A      ?C=0     W
     31       GOYES    End
     1B00000  D0=(5)   00000
     142      A=DAT0   A
     A7E      C=C-1    W
     6DEF     GOTO     L1
End
```

If the test is true, the instruction takes 32 or 33 cycles depending if its address is even or odd. If the test is false, the instruction takes 24 or 25 cycles (the field in question is field **W**; q is 16 nibbles).

`D0=(5)`	`00000`	:10 or 11 cycles.
`A=DAT0`	`A`	: 23 or 24 cycles (reading from even address).
`C=C-1`	`W`	: 20 or 21 cycles.
`GOTO`	`L1`	: 14 cycles.

There are 32 or 33 if the loop is not executed (`C=0 W`) and 93 otherwise (if an instruction with an odd length begins on an even address, the next instruction will begin on an odd address and vice versa).

Moves

Immediate

You may move immediate values into certain registers. There are special instructions for moving zero into a register. Here is a list of possible moves:

- For register **A**:
  - Set field **A** to zero:

    `A=0` `A` (8) `D0`
  - Set any other field to zero:

    `A=0` `b` (4.5+q) `Ab0`
  - Set bit x to zero. The bit number must be from 0 to F. Thus, this instruction can only have effect on the first 4 nibbles of **A**:

    `ABIT=0` x (7.5) `8084`x
  - Set bit x to one. This is the inverse of the previous instruction.

    `ABIT=1` x (7.5) `8085`x
  - Move a value into **A**. This instruction moves $x+1$ nibbles into the register (nibbles $h_0 \ldots h_x$), using the value of **P**: Nibble h_0 is moved into nibble ***P*** of **A**; h_1 is moved into nibble ***P****+1*, etc. Remember that the processor reverses the nibbles moved.

    `LAHEX(`x`)` $h_x \ldots h_0$ (5+q+(5+q)/2) `8082`$xh_0 \ldots h_x$
- For register **B**:
  - Set field **A** to zero:

    `B=0` `A` (8) `D1`
  - Set any other field to zero:

    `B=0` `b` (4.5+q) `Ab1`
- For register **C**:
  - Set field **A** to zero:

    `C=0` `A` (8) `D2`
  - Set any other field to zero:

    `C=0` `b` (4.5+q) `Ab2`
  - Clear bit x (0h $\leq x \leq$ Fh):

    `CBIT=0` x (7.5) `8088`x
  - Set bit x (0h $\leq x \leq$ Fh):

    `CBIT=1` x (7.5) `8089`x
  - Move a value into C:

    `LCHEX` `#h`$x \ldots h_0$ (2+q+(2+q)/2) `3`$xh_0 \ldots h_x$

- For register **D**:
  - Set field **A** to zero:

D=0	A	(8)	D3

  - Set any other field to zero:

D=0	b	(4.5+q)	Ab3

- For register **P**:
  - Move the value *n* (0h ≤ *n* ≤ Fh) into **P**:

P=	*n*	(3)	2*n*

- For register **D0**:
  - Move a value into the 2 least significant nibbles:

D0=(2)	qp	(6)	19pq

  - Move a value into the 4 least significant nibbles:

D0=(4)	srqp	(9)	1Apqrs

  - Move a value into **D0**:

D0=(5)	tsrqp	(10.5)	1Bpqrst

- For register **D1**:
  - Move a value into the 2 least significant nibbles:

D1=(2)	qp	(6)	1Dpq

  - Move a value into the 4 least significant nibbles:

D1=(4)	srqp	(9)	1Epqrs

  - Move a value into **D1**:

D1=(5)	tsrqp	(10.5)	1Fpqrst

- For register **HST**:
  - Clear flag **XM**:

XM=	0	(4.5)	821

  - Clear flag **SB**:

SB=	0	(4.5)	822

  - Clear flag **SR**:

SR=	0	(4.5)	824

  - Clear flag **MP**:

MP=	0	(4.5)	828

  - Clear all four flags:

CLRHST		(4.5)	82F

- For register **ST**:
  - Clear flag *d* (0h ≤ *d* ≤ Fh):

ST=0	*d*	(5.5)	84*d*

  - Clear all flags:

CLRST		(7)	08

  - Set flag *d*:

ST=1	*d*	(5.5)	85*d*

Moving Values

- For Register **A**:
  - Move field **A** of **B** into field **A**:

A=B	A	(8)	D4

  - Move field b of **B** into field b:

A=B	b	(4.5+q)	Ab4

  - The same instructions exist for **C**:

A=C	A	(8)	DA
A=C	b	(4.5+q)	AbA

- For Register **B**:
  - Move field **A** of **A** into field **A**:

B=A	A	(8)	D8

  - Move field b of **A** into field b:

B=A	b	(4.5+q)	Ab8

  - The same instructions exist for **C**:

B=C	A	(8)	D5
B=C	b	(4.5+q)	Ab5

- For Register **C**:
  - Move field **A** of **A** into field **A**:

C=A	A	(8)	D6

  - Move field b of **A** into field b:

C=A	b	(4.5+q)	Ab6

  - The same instructions exist for **B**:

C=B	A	(8)	D9
C=B	b	(4.5+q)	Ab9

  - The same instructions exist for **D**:

C=D	A	(8)	DB
C=D	b	(4.5+q)	AbB

  - Move **P** into nibble *n*:

C=P	*n*	(8)	80C*n*

  - Move flags 0 to 11 of **ST** into field **X**:

C=ST		(7)	09

- For Register **D**:
  - Move field **A** of **C** into field **A**:

D=C	A	(8)	D7

  - Move field b of **C** into field b:

D=C	b	(4.5+q)	Ab7

- For Register **P**:
  - Move nibble *n* of **C** into **P**:

P=C	*n*	(8)	80D*n*

- For Register **D0**:
  - Move field **A** of **A** into **D0**:

D0=A	(9.5)	130

  - Move nibbles 0 to 3 of **A** into **D0**:

D0=AS	(8.5)	138

  - The same instructions exist for **C**:

D0=C	(9.5)	134
D0=CS	(8.5)	13C

- For Register **D1**:
  - Move field **A** of **A** into **D1**:

D1=A	(9.5)	131

  - Move nibbles 0 to 3 of **A** into **D1**:

D1=AS	(8.5)	139

  - The same instructions exist for **C**:

D1=C	(9.5)	135
D1=CS	(8.5)	13D

- For Register **ST**:
  - Move field **X** of **C** into flags 0 to 11 of **ST**:

ST=C	(7)	0A

Saving and Restoring (Rn and RSTK)

- For Register **A**:
  - Save the entire register:

R0=A		(20.5)	100
R1=A		(20.5)	101
R2=A		(20.5)	102
R3=A		(20.5)	103
R4=A		(20.5)	104

  - Save field **A** only:

R0=A	A	(14)	81AF00
R1=A	A	(14)	81AF01
R2=A	A	(14)	81AF02
R3=A	A	(14)	81AF03
R4=A	A	(14)	81AF04

  - Save field a only:

R0=A	a	(9+q)	81Aa00
R1=A	a	(9+q)	81Aa01
R2=A	a	(9+q)	81Aa02
R3=A	a	(9+q)	81Aa03
R4=A	a	(9+q)	81Aa04

  - Restore the entire register:

A=R0		(20.5)	110
A=R1		(20.5)	111
A=R2		(20.5)	112
A=R3		(20.5)	113
A=R4		(20.5)	114

  - Restore field **A** only:

A=R0	A	(14)	81AF10
A=R1	A	(14)	81AF11
A=R2	A	(14)	81AF12
A=R3	A	(14)	81AF13
A=R4	A	(14)	81AF14

  - Restore field a only:

A=R0	a	(9+q)	81Aa10
A=R1	a	(9+q)	81Aa11
A=R2	a	(9+q)	81Aa12
A=R3	a	(9+q)	81Aa13
A=R4	a	(9+q)	81Aa14

- For Register **C**:
  - Save the entire register:

R0=C		(20.5)	108
R1=C		(20.5)	109
R2=C		(20.5)	10A
R3=C		(20.5)	10B
R4=C		(20.5)	10C

  - Save field **A** only:

R0=C	A	(14)	81AF08
R1=C	A	(14)	81AF09
R2=C	A	(14)	81AF0A
R3=C	A	(14)	81AF0B
R4=C	A	(14)	81AF0C

  - Save field a only:

R0=C	a	(9+q)	81Aa08
R1=C	a	(9+q)	81Aa09
R2=C	a	(9+q)	81Aa0A
R3=C	a	(9+q)	81Aa0B
R4=C	a	(9+q)	81Aa0C

  - Restore the entire register:

C=R0		(20.5)	118
C=R1		(20.5)	119
C=R2		(20.5)	11A
C=R3		(20.5)	11B
C=R4		(20.5)	11C

  - Restore field **A** only:

C=R0	A	(14)	81AF18
C=R1	A	(14)	81AF19
C=R2	A	(14)	81AF1A
C=R3	A	(14)	81AF1B
C=R4	A	(14)	81AF1C

  - Restore field a only:

C=R0	a	(9+q)	81Aa18
C=R1	a	(9+q)	81Aa19
C=R2	a	(9+q)	81Aa1A
C=R3	a	(9+q)	81Aa1B
C=R4	a	(9+q)	81Aa1C

  - Restore field **A** from **RSTK**:

C=RSTK		(9)	07

  - Save field **A** into **RSTK**:

RSTK=C		(9)	06

Reading and Writing to Memory

- For Register **A**:
  - Move the data pointed to by D0 into field **A**:
    `A=DAT0     A       (20.5, 3.5)        142`
  - Same for field **B**:
    `A=DAT0     B       (19.5)             14A`
  - Same for field a:
    `A=DAT0     a       (20+q, (q+2)/2)    152a`
  - Same for *x+1* nibbles:
    `A=DAT0     x+1     (19+q, (q+2)/2)    15Ax`
  - The same instructions exist for **D1**:
    `A=DAT1     A       (20.5, 3.5)        143`
    `A=DAT1     B       (19.5)             14B`
    `A=DAT1     a       (20+q, (q+2)/2)    153a`
    `A=DAT1     x+1     (19+q, (q+2)/2)    15Bx`
  - Move field **A** into the address pointed to by **D0**:
    `DAT0=A     A       (19.5)             140`
  - Same for field **B**:
    `DAT0=A     B       (16.5)             148`
  - Same for field a:
    `DAT0=A     a       (19+q)             150a`
  - Same for x+1 nibbles:
    `DAT0=A     x+1     (18+q)             1993`
  - The same instructions exist for **D1**:
    `DAT1=A     A       (19.5)             141`
    `DAT1=A     B       (16.5)             149`
    `DAT1=A     a       (19+q)             151a`
    `DAT1=A     x+1     (18+q)             159x`
- For Register **C**:
  - Move the data pointed to by D0 into field **A**:
    `C=DAT0     A       (20.5, 3.5)        146`
  - Same for field **B**:
    `C=DAT0     B       (19.5)             14E`
  - Same for field a:
    `C=DAT0     a       (20+q, (q+2)/2)    156a`
  - Same for x+1 nibbles:
    `C=DAT0     x+1     (19+q, (q+2)/2)    15Ex`

- The same instructions exist for **D1**:

C=DAT1	A	(20.5, 3.5)	147
C=DAT1	B	(19.5)	14F
C=DAT1	a	(20+q, (q+2)/2)	157a
C=DAT1	*x*+1	(19+q, (q+2)/2)	15F*x*

- Move field **A** into the address pointed to by **D0**:

DAT0=C	A	(19.5)	144

- Same for field **B**:

DAT0=C	B	(16.5)	14C

- Same for field a:

DAT0=C	a	(19+q)	154a

- Same for x+1 nibbles:

DAT0=C	*x*+1	(18+q)	15C*x*

- The same instructions exist for **D1**:

DAT1=C	A	(19.5)	145
DAT1=C	B	(16.5)	14D
DAT1=C	a	(19+q)	155a
DAT1=C	*x*+1	(18+q)	15D*x*

Input and Output

The following instructions are for reading the keyboard as well as using the HP 48's speaker (see **Chapter 9**). Caution: The instructions A=IN and C=IN corrupt the memory area read when used in RAM (see **Chapter 9**).

- For Register **A**:
  - Read the Input (into nibbles 0,1,2 and 3 of A):

A=IN	(8.5)	802

- For Register **C**:
  - Read the Input (into nibbles 0,1,2 and 3 of C):

C=IN	(8.5)	803

  - Write field **X** to the output:

OUT=C	(7.5)	801

  - Write nibble 0 into nibble 0 of the output register:

OUT=CS	(5.5)	800

Exchanging Register Contents

- For Register **A**:
  - Exchange field **A** with field **A** of **B**:

ABEX	A	(8)	DC

  - Exchange field b with field b of **B**:

ABEX	b	(4.5+q)	AbC

  - The same instructions exist for **C**:

ACEX	A	(8)	DE
ACEX	b	(4.5+q)	AbE

  - Exchange with **R0**:

AR0EX		(20.5)	120

  - Exchange field **A** with field **A** of **R0**:

AR0EX	A	(14)	81AF20

  - Exchange field a with field a of **R0**:

AR0EX	a	(9+q)	81Aa20

  - The same instructions exist for **R1**:

AR1EX		(20.5)	121
AR1EX	A	(14)	81AF21
AR1EX	a	(9+q)	81Aa21

  - The same instructions exist for **R2**:

AR2EX		(20.5)	122
AR2EX	A	(14)	81AF22
AR2EX	a	(9+q)	81Aa22

  - The same instructions exist for **R3**:

AR3EX		(20.5)	123
AR3EX	A	(14)	81AF23
AR3EX	a	(9+q)	81Aa23

  - The same instructions exist for **R4**:

AR4EX		(20.5)	124
AR4EX	A	(14)	81AF24
AR4EX	a	(9+q)	81Aa24

  - Exchange field **A** with **D0**:

AD0EX	(9.5)	132

  - Exchange nibbles 0 to 3 with those of **D0**:

AD0XS	(8.5)	13A

  - The same instructions exist for **D1**:

AD1EX	(9.5)	133
AD1XS	(8.5)	13B

- For register **B**:
  - Exchange field **A** with field **A** of **A**:

    ```
    BAEX    A    (8)       DC
    ```
  - Exchange field b with field b of **A**:

    ```
    BAEX    b    (4.5+q)   AbC
    ```
  - The same instructions exist for **C**:

    ```
    BCEX    A    (8)       DD
    BCEX    b    (4.5+q)   AbD
    ```
- For Register **C**:
  - Exchange field **A** with field **A** of **A**:

    ```
    CAEX    A    (8)       DE
    ```
  - Exchange field b with field b of **A**:

    ```
    CAEX    b    (4.5+q)   AbE
    ```
  - The same instructions exist for **B**:

    ```
    CBEX    A    (8)       DD
    CBEX    b    (4.5+q)   AbD
    ```
  - The same instructions exist for **D**:

    ```
    CDEX    A    (8)       DF
    CDEX    b    (4.5+q)   AbF
    ```
  - Exchange with **R0**:

    ```
    CR0EX        (20.5)    128
    ```
  - Exchange field **A** with field **A** of **R0**:

    ```
    CR0EX   A    (14)      81AF28
    ```
  - Exchange field a with field a of **R0**:

    ```
    CR0EX   a    (9+q)     81Aa28
    ```
  - The same instructions exist for **R1**:

    ```
    CR1EX        (20.5)    129
    CR1EX   A    (14)      81AF29
    CR1EX   a    (9+q)     81Aa29
    ```
  - The same instructions exist for **R2**:

    ```
    CR2EX        (20.5)    12A
    CR2EX   A    (14)      81AF2A
    CR2EX   a    (9+q)     81Aa2A
    ```
  - The same instructions exist for **R3**:

    ```
    CR3EX        (20.5)    12B
    CR3EX   A    (14)      81AF2B
    CR3EX   a    (9+q)     81Aa2B
    ```

- The same instructions exist for **R4**:

CR4EX		(20.5)	12C
CR4EX	A	(14)	81AF2C
CR4EX	a	(9+q)	81Aa2C

- Exchange field **A** with **D0**:

CD0EX		(9.5)	136

- Exchange nibbles 0 to 3 with those of **D0**:

CD0XS		(8.5)	13E

- The same instructions exist for **D1**:

CD1EX		(9.5)	137
CD1XS		(8.5)	13F

- Exchange nibble *n* with **P**.

CPEX	*n*	(8)	80F*n*

- Exchange field **X** with flags 0 to 11 of **ST**.

CSTEX		(7)	0B

• For register **D**:

- Exchange field **A** with field **A** of **C**.

DCEX	A	(8)	DF

- Exchange field b with field b of **C**.

DCEX	b	(4.5+q)	AbF

Arithmetic Operations

Increment

These instructions modify the value of the **CARRY** flag.

- For register **A**:
  - Increment field **A**:

A=A+1	A	(8)	E4

  - Increment field a:

A=A+1	a	(4.5+q)	Ba4

  - Increment field **A** by x+1 (0h ≦ x ≦ Fh):

A=A+x+1	A	(13)	818F0x

  - Increment field a by x+1:

A=A+x+1	a	(8+q)	818a0x

- For register **B**:
  - Increment field **A**:

B=B+1	A	(8)	E5

  - Increment field a:

B=B+1	a	(4.5+q)	Ba5

  - Increment field **A** by x+1 (0h ≦ x ≦ Fh):

B=B+x+1	A	(13)	818F1x

  - Increment field a by x+1:

B=B+x+1	a	(8+q)	818a1x

- For register **C**:
  - Increment field **A**:

C=C+1	A	(8)	E6

  - Increment field a:

C=C+1	a	(4.5+q)	Ba6

  - Increment field **A** by x+1 (0h ≦ x ≦ Fh):

C=C+x+1	A	(13)	818F2x

  - Increment field a by x+1:

C=C+x+1	a	(8+q)	818a2x

- For register **D**:
  - Increment field **A**:

D=D+1	A	(8)	E7

  - Increment field a:

D=D+1	a	(4.5+q)	Ba7

- Increment field **A** by x+1 (0h ≦ x ≦ Fh):
  D=D+x+1 A (13) 818F3x
- Increment field a by x+1:
  D=D+x+1 a (8+q) 818a3x

• For register **P**:
- Increment:
  P=P+1 (4) 0C

• For register **D0**:
- Increment by x+1:
  D0=D0+ x+1 (8.5) 16x

• For register **D1**:
- Increment by x+1:
  D1=D1+ x+1 (8.5) 17x

Addition

These instructions modify the value of the **CARRY** flag.

• For register **A**:
- Add field **A** of **B** to field **A**:
  A=A+B A (8) C0
- Add field a of **B** to field a:
  A=A+B a (4.5+q) Aa0
- The same instructions exist for **C**:
  A=A+C A (8) CA
  A=A+C a (4.5+q) AaA

• For register **B**:
- Add field **A** of **A** to field **A**:
  B=B+A A (8) C8
- Add field a of **A** to field a:
  B=B+A a (4.5+q) Aa8
- The same instructions exist for **C**:
  B=B+C A (8) C1
  B=B+C a (4.5+q) Aa1

• For Register **C**:
- Add field **A** of **A** to field **A**:
  C=C+A A (8) C2
- Add field a of **A** to field a:
  C=C+A a (4.5+q) Aa2

- The same instructions exist for **B**:

C=C+B	A	(8)	C9
C=C+B	a	(4.5+q)	Aa9

- The same instructions exist for **D**:

C=C+D	A	(8)	CB
C=C+D	a	(4.5+q)	AaB

- Add P+1 to field **A**:

C+P+1		(9.5)	809

- For register **D**:
  - Add field **A** of **C** to field **A**:

D=D+C	A	(8)	C3

  - Add field a of **C** to field a:

D=D+C	a	(4.5+q)	Aa3

Decrement

These instructions modify the value of the **CARRY** flag.

- For register **A**:
  - Decrement field **A**:

A=A-1	A	(8)	CC

  - Decrement field a:

A=A-1	a	(4.5+q)	AaC

  - Decrement field **A** by x+1 (0h ≤ x ≤ Fh):

A=A-(x+1)	A	(13)	818F8x

  - Decrement field a by x+1:*

A=A-(x+1)	a	(8+q)	818a8x

- For register **B**:
  - Decrement field **A**:

B=B-1	A	(8)	CD

  - Decrement field a:

B=B-1	a	(4.5+q)	AaD

  - Decrement field **A** by x+1 (0h ≤ x ≤ Fh):

B=B-(x+1)	A	(13)	818F9x

  - Decrement field a by x+1:*

B=B-(x+1)	a	(8+q)	818a9x

*Caution: This instruction does not work correctly except for fields**X**, **M**, **B**, and **W**.

- For register **C**:
  - Decrement field **A**:
    `C=C-1` A (8) `CE`
  - Decrement field a:
    `C=C-1` a (4.5+q) `AaE`
  - Decrement field **A** by x+1 (0h ≤ x ≤ Fh):
    `C=C-(x+1)` A (13) `818FAx`
  - Decrement field a by x+1:*
    `C=C-(x+1)` a (8+q) `818aAx`
- For register **D**:
  - Decrement field **A**:
    `D=D-1` A (8) `CF`
  - Decrement field a:
    `D=D-1` a (4.5+q) `AaF`
  - Decrement field **A** by x+1 (0h ≤ x ≤ Fh):
    `D=D-(x+1)` A (13) `818FBx`
  - Decrement field a by x+1:*
    `D=D-(x+1)` a (8+q) `818aBx`
- For register **P**:
  - Decrement:
    `P=P-1` (4) `0D`
- For register **D0**:
  - Decrement by x+1:
    `D0=D0-` x+1 (8.5) `18x`
- For register **D1**:
  - Decrement by x+1:
    `D1=D1-` x+1 (8.5) `1Cx`

Subtraction

These instructions modify the value of the **CARRY** flag.

- For register **A**:
  - Subtract field **A** of **C** from field **A**:
    `A=A-C` A (8) `EA`
  - Subtract field a of **C** from field a:
    `A=A-C` a (4.5+q) `BaA`

*Caution: This instruction does not work correctly except for fields **X**, **M**, **B**, and **W**.

- Subtract field **A** from field **A** of **B** storing the result in field **A**:
  A=B-A   A   (8)   EC
- Subtract field a from field a of **B** storing the result in field a:
  A=B-A   a   (4.5+q)   BaC

- For register **B**:
  - Subtract field **A** of **A** from field **A**:
    B=B-A   A   (8)   E8
  - Subtract field a of **A** from field a:
    B=B-A   a   (4.5+q)   Ba8
  - These same instructions exist for **C**:
    B=B-C   A   (8)   E1
    B=B-C   a   (4.5+q)   Ba1
  - Subtract field **A** from field **A** of **C** storing the result in field **A**:
    B=C-B   A   (8)   ED
  - Subtract field a from field a of **C** storing the result in field a:
    B=C-B   a   (4.5+q)   BaD
- For Register **C**:
  - Subtract field **A** of **A** from field **A**:
    C=C-A   A   (8)   E2
  - Subtract field a of **A** from field a:
    C=C-A   a   (4.5+q)   Ba2
  - These same instructions exist for **D**:
    C=C-D   A   (8)   EB
    C=C-D   a   (4.5+q)   BaB
  - Subtract field **A** from field **A** of **A** storing the result in field **A**:
    C=A-C   A   (8)   EE
  - Subtract field a from field a of **A** storing the result in field a:
    C=A-C   a   (4.5+q)   BaE
- For register **D**:
  - Subtract field **A** of **C** from field **A**:
    D=D-C   A   (8)   E3
  - Subtract field a of **C** from field a:
    D=D-C   a   (4.5+q)   Ba3
  - Subtract field **A** from field **A** of **C** storing the result in field **A**:
    D=C-D   A   (8)   EF
  - Subtract field a from field a of **C** storing the result in field a:
    D=C-D   a   (4.5+q)   BaF

Logical AND

- For register **A**:
  - Between field **A** and field **A** of **B**:

A=A&B	A	(11)	0EF0

  - Between field a and field a of **B**:

A=A&B	a	(6+q)	0Ea0

  - The same instructions exist for **C**:

A=A&C	A	(11)	0EF6
A=A&C	a	(6+q)	0Ea6

- For register **B**:
  - Between field **A** and field **A** of **A**:

B=B&A	A	(11)	0EF4

  - Between field a and field a of **A**:

B=B&A	a	(6+q)	0Ea4

  - The same instructions exist for **C**:

B=B&C	A	(11)	0EF1
B=B&C	a	(6+q)	0Ea1

- For register **C**:
  - Between field **A** and field **A** of **A**:

C=C&A	A	(11)	0EF2

  - Between field a and field a of **A**:

C=C&A	a	(6+q)	0Ea2

  - The same instructions exist for **B**:

C=C&B	A	(11)	0EF5
C=C&B	a	(6+q)	0Ea5

  - The same instructions exist for **D**:

C=C&D	A	(11)	0EF7
C=C&D	a	(6+q)	0Ea7

- For register **D**:
  - Between field **A** and field **A** of **C**:

D=D&C	A	(11)	0EF3

  - Between field a and field a of **C**:

D=D&C	a	(6+q)	0Ea3

Logical OR

- For register **A**:
  - Between field **A** and field **A** of **B**:

A=A!B	A	(11)	0EF8

  - Between field a and field a of **B**:

A=A!B	a	(6+q)	0Ea8

  - The same instructions exist for **C**:

A=A!C	A	(11)	0EFE
A=A!C	a	(6+q)	0EaE

- For register **B**:
  - Between field **A** and field **A** of **A**:

B=B!A	A	(11)	0EFC

  - Between field a and field a of **A**:

B=B!A	a	(6+q)	0EaC

  - The same instructions exist for **C**:

B=B!C	A	(11)	0EF9
B=B!C	a	(6+q)	0Ea9

- For register **C**:
  - Between field **A** and field **A** of **A**:

C=C!A	A	(11)	0EFA

  - Between field a and field a of **A**:

C=C!A	a	(6+q)	0EaA

  - The same instructions exist for **B**:

C=C!B	A	(11)	0EFD
C=C!B	a	(6+q)	0EaD

  - The same instructions exist for **D**:

C=C!D	A	(11)	0EFF
C=C!D	a	(6+q)	0EaF

- For register **D**:
  - Between field **A** and field **A** of **C**:

D=D!C	A	(11)	0EFB

  - Between field a and field a of **C**:

D=D!C	a	(6+q)	0EaB

Logical NOT

These instructions modify the value of the **CARRY** flag.

- For register **A**:
  - On field **A**:

A=-A-1	A	(8)	FC

  - On field b:

A=-A-1	b	(4.5+q)	BbC

- For register **B**:
  - On field **A**:

B=-B-1	A	(8)	FD

  - On field b:

B=-B-1	b	(4.5+q)	BbD

- For register **C**:
  - On field **A**:

C=-C-1	A	(8)	FE

  - On field b:

C=-C-1	b	(4.5+q)	BbE

- For register **D**:
  - On field **A**:

D=-D-1	A	(8)	FF

  - On field b:

D=-D-1	b	(4.5+q)	BbF

2's Complement

These instructions modify the value of the **CARRY** flag.

- For register **A**:
  - On field **A**:

A=-A	A	(8)	F8

  - On field b:

A=-A	b	(4.5+q)	Bb8

- For register **B**:
  - On field **A**:

B=-B	A	(8)	F9

  - On field b:

B=-B	b	(4.5+q)	Bb9

- For register **C**:
  - On field **A**:

C=-C	A	(8)	FA

  - On field b:

C=-C	b	(4.5+q)	BbA

- For register **D**:
  - On field **A**:

D=-D	A	(8)	FB

  - On field b:

D=-D	b	(4.5+q)	BbB

Multiplying by 2

- For register **A**:
  - Multiply field **A** by 2:

A=A+A	A	(8)	C4

  - Multiply field a by 2:

A=A+A	a	(4.5+q)	Aa4

- For register **B**:
  - Multiply field **A** by 2:

B=B+B	A	(8)	C5

  - Multiply field a by 2:

B=B+B	a	(4.5+q)	Aa5

- For register **C**:
  - Multiply field **A** by 2:

C=C+C	A	(8)	C6

  - Multiply field a by 2:

C=C+C	a	(4.5+q)	Aa6

- For register **D**:
  - Multiply field **A** by 2:

D=D+D	A	(8)	C7

  - Multiply field a by 2:

D=D+D	a	(4.5+q)	Aa7

Dividing by 2

This operation is performed by shifting the register right one bit. The bit shifted out (least significant) is lost, but **SB** is set if it was non-null (you must do an `SB=0` first), and the bit shifted in (most significant) is always zero.

- For register **A**:
  - Divide by 2:

ASRB		(21.5)	81C
  - Divide field **A** by 2:

ASRB	A	(13.5)	819F0
  - Divide field a by 2:

ASRB	a	(8.5+q)	819a0
- For register **B**:
  - Divide by 2:

BSRB		(21.5)	81D
  - Divide field **A** by 2:

BSRB	A	(13.5)	819F1
  - Divide field a by 2:

BSRB	a	(8.5+q)	819a1
- For register **C**:
  - Divide by 2:

CSRB		(21.5)	81E
  - Divide field **A** by 2:

CSRB	A	(13.5)	819F2
  - Divide field a by 2:

CSRB	a	(8.5+q)	819a2
- For register **D**:
  - Divide by 2:

DSRB		(21.5)	81F
  - Divide field **A** by 2:

DSRB	A	(13.5)	819F3
  - Divide field a by 2:

DSRB	a	(8.5+q)	819a3

Multiplying by 16

This operation shifts the register left one nibble. The nibble shifted out (most significant) is lost, but **SB** is set if it was non-null (you must do an SB=0 first), and the nibble shifted in (least significant) is always zero.

- For register **A**:
  - Multiply field **A** by 16:

    ```
    ASL     A     (9)       F0
    ```
  - Multiply field b by 16:

    ```
    ASL     b     (5.5+q)   Bb0
    ```
- For register **B**:
  - Multiply field **A** by 16:

    ```
    BSL     A     (9)       F1
    ```
  - Multiply field b by 16:

    ```
    BSL     b     (5.5+q)   Bb1
    ```
- For register **C**:
  - Multiply field **A** by 16:

    ```
    CSL     A     (9)       F2
    ```
  - Multiply field b by 16:

    ```
    CSL     b     (5.5+q)   Bb2
    ```
- For register **D**:
  - Multiply field **A** by 16:

    ```
    DSL     A     (9)       F3
    ```
  - Multiply field b by 16:

    ```
    DSL     b     (5.5+q)   Bb3
    ```

Dividing by 16

This operation shifts the register right one nibble. The nibble shifted out (least significant) is lost, but **SB** is set if it was non-null (you must do an SB=0 first), and the nibble shifted in (most significant) is always zero.

- For register **A**:
  - Divide field **A** by 16:

    ```
    ASR     A     (9)       F4
    ```
  - Divide field b by 16:

    ```
    ASR     b     (5.5+q)   Bb4
    ```

- For register **B**:
  - Divide field **A** by 16:

BSR	A	(9)	F5

  - Divide field b by 16:

BSR	b	(5.5+q)	Bb5

- For register **C**:
  - Divide field **A** by 16:

CSR	A	(9)	F6

  - Divide field b by 16:

CSR	b	(5.5+q)	Bb6

- For register **D**:
  - Divide field **A** by 16:

DSR	A	(9)	F7

  - Divide field b by 16:

DSR	b	(5.5+q)	Bb7

Rotating Left (one nibble)

This operation performs a left circular rotation of the register by nibbles. Nibble 0h is moved to 1h, 1h is moved to 2h, etc. The most significant nibble is moved to the least significant nibble position. SLC stands for "Shift Left Circular."

- For register **A**:

ASLC	(22.5)	810

- For register **B**:

BSLC	(22.5)	811

- For register **C**:

CSLC	(22.5)	812

- For register **D**:

DSLC	(22.5)	813

Rotating Right (one nibble)

This operation performs a right circular rotation of the register by nibbles. Nibble Fh is moved to Eh, Eh is moved to Dh, etc. The least significant nibble is moved to the most significant nibble position. SRC stands for "Shift Right Circular."

- For register **A**:

  ASRC (22.5) 814
- For register **B**:

  BSRC (22.5) 815
- For register **C**:

  CSRC (22.5) 816
- For register **D**:

  DSRC (22.5) 817

Jumps

To calculate the distance of relative jumps: Count the number of nibbles from the end of the jump instruction (not including the distance nibbles) to the beginning of the desired instruction. To jump backwards, use the 2's complement of the distance. For a relative GOT O, the code is 6*aaa*, where *aaa* is the jump distance. Thus, to jump between addresses $@_1$ and $@_2$:

- If the jump is forward, $(@_2-(@_1+1))$ calculates the distance. You add 1 to $@_1$ because that's the length of the jump instruction 6*aaa* (you don't count the nibbles *aaa* in the calculation). Thus, if $@_1$=#00123h and $@_2$=#00456h, the distance to jump is 332h nibbles, and is coded as 6233 (don't forget that the microprocessor reverses data).
- If the jump is backward, $((@_1+1)-@_2)$ calculates the distance. Thus, if $@_1$=#00456h and $@_2$=#00123h, the distance to jump is 334h nibbles, coded as 6CCC (the 2's complement of 334h is CCCh).

The limits of these jumps are as follows:

- Using 2 nibbles for the length, you can jump -80h to +7Fh nibbles.
- Using 3 nibbles for the length, -800h to +7FFFh nibbles.
- Using 4 nibbles for the length, -8000h to +7FFFh nibbles.

Note: In assembly program listings, you can use labels to indicate jump addresses without needing to calculate the distance yourself.

Direct relative unconditional

GOTO	abc	(14)	6cba
GOLONG	abcd	(17)	8Cdcba

Direct relative conditional

These jumps depend on the state of the CARR Y flag.

- Jump on **CARRY** clear:

GONC	ab	(12.5/4.5)	5ba

- Jump on **CARRY** set:

GOC	ab	(12.5/4.5)	4ba

Absolute

GOVLNG	abcde	(18.5)	8Dedcba

Register direct

- Using register **A**:
  - Jump to the address contained in field **A**:

    `PC=A` (19) `81B2`
  - Jump to the address contained in field **A**, saving the address of the next instruction into field **A**:

    `APCex` (19) `81B6`
- Using register **C**:
  - Jump to the address contained in field **A**:

    `PC=C` (19) `81B3`
  - Jump to the address contained in field **A**, saving the address of the next instruction into field **A**:

    `CPCex` (19) `81B7`

Register indirect

- Using register **A**:
  - Jump to the address contained in the 5 nibbles pointed to by field **A** (the 5 nibbles are read from the address contained in field **A**, and execution continues at this address):

    `PC=(A)` (26, 3.5) `808C`
- Using register **C**:
  - Jump to the address contained in the 5 nibbles pointed to by field **C**:

    `PC=(C)` (26, 3.5) `808E`

Getting the Program Counter

Jump instructions cause changes to the program counter **PC**. The following instructions allow you to find out exactly what address is contained in the program counter—the address of the next instruction to be executed.

- Move **PC** into field **A** of register **A**:

  `A=PC` (11) `81B4`
- Move **PC** into field **A** of register **C**:

  `C=PC` (11) `81B5`

Calling Subroutines

The distance of a relative subroutine call is calculated dif ferently than for a relative jump. You count from the first nibble of the instruction after the subroutine call. Example:

```
GOSUB @1
@2        (next instruction)
@1        (some useful subroutine)
```

In this program, the distance of the call would be $@_1$-$@_2$. As with jumps, you must use the 2's complement of the distance if $@_1$<$@_2$. (Note: In assembly programs listings, you can use labels to indicate subroutine addresses without needing to calculate the distance yourself.)

Direct Relative Unconditional

GOSUB	abc	(15)	7bca
GOSUBL	abcd	(18)	8Edcba

Absolute

GOSBVL	abcde	(19.5)	8Fedcba

Returning From Subroutines

- Unconditional returns:
  - Simple return:
    RTN (11) 01
  - Return after clearing the **CARRY**:
    RTNCC (11) 03
  - Return after setting the **CARRY**:
    RTNSC (11) 02
  - Return after setting **XM**:
    RTNSXM (11) 00
  - Return from interrupt
    RTI (11) 0F
- Conditional returns:
  - Return if the **CARRY** is set:
    RTNC (12.5/4.5) 400
  - Return if the **CARRY** is clear:
    RTNNC (12.5/4.5) 500

Comparisons

All comparisons are of the form

? *<register> <comparator> <register* or *immediate> <field>*

A comparison instruction will always be followed by a jump (GOYES) or a conditional return from subroutine (RTNYES). The instruction that follows a comparison has the following rules:

- The instruction itself is always 2 nibbles long.
- 00 is RTNYES;
- Anything else is the value of a relative jump GOYES. The jump distance is counted from the address of the GOYES instruction (see **Section IV** for more information on calculating jump distances).

Notes:

- These instructions modify the value of the **CARRY** flag. The **CARRY** is set if the comparison is true.
- These are unsigned comparisons as the register values are positive numbers.

Immediate

- For register **A**:
  - Is field **A** zero?

?A=0	A	(21.5/13.5)	8A8

  - Is field a zero?

?A=0	a	(16.5+q/8.5+q)	9a8

  - Is field **A** non zero?

?A#0	A	(21.5/13.5)	8AC

  - Is field a non zero?

?A#0	a	(16.5+q/8.5+q)	9aC

  - Is bit x (0h ≦ x ≦ Fh) clear?

?ABIT=0	x	(20.5/12.5)	8086x

  - Is bit x (0h ≦ x ≦ Fh) set?

?ABIT=1	x	(20.5/12.5)	8087x

- For register **B**:
  - Is field **A** zero?

    `?B=0` A (21.5/13.5) 8A9
  - Is field a zero?

    `?B=0` a (16.5+q/8.5+q) 9a9
  - Is field **A** non zero?

    `?B#0` A (21.5/13.5) 8AD
  - Is field a non zero?

    `?B#0` a (16.5+q/8.5+q) 9aD
- For register **C**:
  - Is field **A** zero?

    `?C=0` A (21.5/13.5) 8AA
  - Is field a zero?

    `?C=0` a (16.5+q/8.5+q) 9aA
  - Is field **A** non zero?

    `?C#0` A (21.5/13.5) 8AE
  - Is field a non zero?

    `?C#0` a (16.5+q/8.5+q) 9aE
  - Is bit x (0h $\leqq x \leqq$ Fh) clear?

    `?CBIT=0` x (20.5/12.5) 808Ax
  - Is bit x (0h $\leqq x \leqq$ Fh) set?

    `?CBIT=1` x (20.5/12.5) 808Bx
- For register **D**:
  - Is field **A** zero?

    `?D=0` A (21.5/13.5) 8AB
  - Is field a zero?

    `?D=0` a (16.5+q/8.5+q) 9aB
  - Is field **A** non zero?

    `?D#0` A (21.5/13.5) 8AF
  - Is field a non zero?

    `?D#0` a (16.5+q/8.5+q) 9aF
- For register **HST**:
  - Is **XM** clear?

    `?XM=0` (15.5/7.5) 831
  - Is **SB** clear?

    `?SB=0` (15.5/7.5) 832
  - Is **SR** clear?

    `?SR=0` (15.5/7.5) 834
  - Is **MP** clear?

    `?MP=0` (15.5/7.5) 838

- For register **P**:
  - Is **P** equal to *n*?

?P=	*n*	(15.5/7.5)	89*n*

  - Is **P** not equal to *n*?

?P#	*n*	(15.5/7.5)	88*n*

- For register **ST**:
  - Is flag *n* clear?

?ST=0	*n*	(16.5/8.5)	86*n*

  - Is flag *n* set?

?ST=1	*n*	(16.5/8.5)	87*n*

  - Is flag *n* not clear?

?ST#0	*n*	(16.5/8.5)	87*n*

  - Is flag *n* not set?

?ST#1	*n*	(16.5/8.5)	86*n*

Comparing registers

- For register A:
  - Is field **A** equal to field **A** of register **B**?

?A=B	A	(21.5/13.5)	8A0

  - Is field a equal to field a of register **B**?

?A=B	a	(16.5+q/8.5+q)	9a0

  - The same instructions exist for **C**:

?A=C	A	(21.5/13.5)	8A2
?A=C	a	(16.5+q/8.5+q)	9a2

  - Is field **A** not equal to field **A** of register **B**?

?A#B	A	(21.5/13.5)	8A4

  - Is field a not equal to field a of register **B**?

?A#B	a	(16.5+q/8.5+q)	9a4

  - The same instructions exist for **C**:

?A#C	A	(21.5/13.5)	8A6
?A#C	a	(16.5+q/8.5+q)	9a6

  - Is field **A** less than or equal to field **A** of register **B**?

?A<=B	A	(21.5/13.5)	8BC

  - Is field a less than or equal to field a of register **B**?

?A<=B	b	(16.5+q/8.5+q)	9bC

  - Is field **A** less than field **A** of register **B**?

?A<B	A	(21.5/13.5)	8B4

  - Is field a less than field a of register **B**?

?A<B	b	(16.5+q/8.5+q)	9b4

- Is field **A** greater than or equal to field **A** of register **B**?
  `?A>=B        A     (21.5/13.5)        8B8`
- Is field a greater than or equal to field a of register **B**?
  `?A>=B        b     (16.5+q/8.5+q)     9b8`
- Is field **A** greater than field **A** of register **B**?
  `?A>B         A     (21.5/13.5)        8B0`
- Is field a greater than field a of register **B**?
  `?A>B         b     (16.5+q/8.5+q)     9b0`

• For register **B**:

- Is field **A** equal to field **A** of register **A**?
  `?B=A         A     (21.5/13.5)        8A0`
- Is field a equal to field a of register **A**?
  `?B=A         a     (16.5+q/8.5+q)     9a0`
- The same instructions exist for **C**:
  `?B=C         A     (21.5/13.5)        8A1`
  `?B=C         a     (16.5+q/8.5+q)     9a1`
- Is field **A** not equal to field **A** of register **A**?
  `?B#A         A     (21.5/13.5)        8A4`
- Is field a not equal to field a of register **A**?
  `?B#A         a     (16.5+q/8.5+q)     9a4`
- The same instructions exist for **C**:
  `?B#C         A     (21.5/13.5)        8A5`
  `?B#C         a     (16.5+q/8.5+q)     9a5`
- Is field **A** less than or equal to field **A** of register **C**?
  `?B<=C        A     (21.5/13.5)        8BD`
- Is field a less than or equal to field a of register **C**?
  `?B<=C        b     (16.5+q/8.5+q)     9bD`
- Is field **A** less than field **A** of register **C**?
  `?B<C         A     (21.5/13.5)        8B5`
- Is field a less than field a of register **C**?
  `?B<C         b     (16.5+q/8.5+q)     9b5`
- Is field **A** greater than or equal to field **A** of register **C**?
  `?B>=C        A     (21.5/13.5)        8B9`
- Is field a greater than or equal to field a of register **C**?
  `?B>=C        b     (16.5+q/8.5+q)     9b9`
- Is field **A** greater than field **A** of register **C**?
  `?B>C         A     (21.5/13.5)        8B1`
- Is field a greater than field a of register **C**?
  `?B>C         b     (16.5+q/8.5+q)     9b1`

- For register **C**:
  - Is field **A** equal to field **A** of register **A**?
    `?C=A    A    (21.5/13.5)    8A2`
  - Is field a equal to field a of register **A**?
    `?C=A    a    (16.5+q/8.5+q)    9a2`
  - The same instructions exist for **B**:
    `?C=B    A    (21.5/13.5)    8A1`
    `?C=B    a    (16.5+q/8.5+q)    9a1`
  - The same instructions exist for **D**:
    `?C=D    A    (21.5/13.5)    8A3`
    `?C=D    a    (16.5+q/8.5+q)    9a3`
  - Is field **A** not equal to field **A** of register **A**?
    `?C#A    A    (21.5/13.5)    8A6`
  - Is field a not equal to field a of register **A**?
    `?C#A    a    (16.5+q/8.5+q)    9a6`
  - The same instructions exist for **B**:
    `?C#B    A    (21.5/13.5)    8A5`
    `?C#B    a    (16.5+q/8.5+q)    9a5`
  - The same instructions exist for **D**:
    `?C#D    A    (21.5/13.5)    8A7`
    `?C#D    a    (16.5+q/8.5+q)    9a7`
  - Is field **A** less than or equal to field **A** of register **A**?
    `?C<=A    A    (21.5/13.5)    8BE`
  - Is field a less than or equal to field a of register **C**?
    `?C<=A    b    (16.5+q/8.5+q)    9bE`
  - Is field **A** less than field **A** of register **A**?
    `?C<A    A    (21.5/13.5)    8B6`
  - Is field a less than field a of register **A**?
    `?C<A    b    (16.5+q/8.5+q)    9b6`
  - Is field **A** greater than or equal to field **A** of register **A**?
    `?C>=A    A    (21.5/13.5)    8BA`
  - Is field a greater than or equal to field a of register **A**?
    `?C>=A    b    (16.5+q/8.5+q)    9bA`
  - Is field **A** greater than field **A** of register **A**?
    `?C>A    A    (21.5/13.5)    8B2`
  - Is field a greater than field a of register **A**?
    `?C>A    b    (16.5+q/8.5+q)    9b2`

- For register **D**:
  - Is field **A** equal to field **A** of register **C**?

    ?D=C A (21.5/13.5) 8A3
  - Is field a equal to field a of register **A**?

    ?D=C a (16.5+q/8.5+q) 9a3
  - Is field **A** not equal to field **A** of register **C**?

    ?D#C A (21.5/13.5) 8A7
  - Is field a not equal to field a of register **C**?

    ?D#C a (16.5+q/8.5+q) 9a7
  - Is field **A** less than or equal to field **A** of register **C**?

    ?D<=C A (21.5/13.5) 8BF
  - Is field a less than or equal to field a of register **C**?

    ?D<=C b (16.5+q/8.5+q) 9bF
  - Is field **A** less than field **A** of register **C**?

    ?D<C A (21.5/13.5) 8B7
  - Is field a less than field a of register **C**?

    ?D<C b (16.5+q/8.5+q) 9b7
  - Is field **A** greater than or equal to field **A** of register **C**?

    ?D>=C A (21.5/13.5) 8BB
  - Is field a greater than or equal to field a of register **C**?

    ?D>=C b (16.5+q/8.5+q) 9bB
  - Is field **A** greater than field **A** of register **C**?

    ?D>C A (21.5/13.5) 8B3
  - Is field a greater than field a of register **C**?

    ?D>C b (16.5+q/8.5+q) 9b3

Bus Commands

These commands are not well known because there is little documentation in the HP 71 HDS published by Hewlett-Packard.

- Commands:
  - Command "B":
    `BUSCB` (10) 8083
  - Command "C":
    `BUSCC` (8.5) 80B
  - Command "D":
    `BUSCD` (10) 808D
  - UN configure all chips on the bus:
    `RESET` (7.5) 80A
  - Shutdown all chips on the bus:
    `SHUTDN` (6.5) 807
  - Un-configure the module found at the address contained in field A of register C:
    `UNCNFG` (14.5) 804
  - Copy field A of register C into the configuration register of the current module (the first module not configured on the bus). This command is generally executed just after an `UNCNFG`. These two commands allow access to the hidden ROM by displacing the user RAM (see the chapters on memory). Memories of 32 Kb or more need a double configuration. The first is the 2's complement of the module size (#100000 - the size in nibbles), which permits use of only one part of the module. The second is the starting address. Thus the displacement of internal RAM from #70000h to #F0000 is done by an `UNCNFG` on #70000h, then by a double `CONFIG` on #F0000h. Returning to normal mode would be done by an `UNCNFG` on #F0000h, followed by `CONFIG` on #F0000h, then on #70000h.
    `CONFIG` (13.5) 805
- Get the identification of the current module. The identifier is stored in field A of register C.
  `C=ID` (13.5) 806
- Find the service requested by a module on the bus. The result is stored in nibble 0 of register C, 1 bit for each type of request.
  `SREQ?` (9.5) 80E

Control Instructions

- Interrupt control instructions:
  - Enable maskable interrupts:
    `INTON` (7) `8080`
  - Disable maskable interrupts:
    `INTOFF` (7) `808F`
  - Clear all interrupts:
    `RSI` (8.5) `80810`
- These instructions change the calculation mode for mathematical operations as described in **Chapter 2**:
  - Set mode to decimal:
    `SETDEC` (4) `05`
  - Set mode to hexadecimal:
    `SETHEX` (4) `04`

NOPs (Instructions with No Effect)

In order to save room in a machine language program for future additions, NOP instructions may be inserted. The three following jump instructions are commonly used as such:

NOP3	420
NOP4	6300
NOP5	64000

Pseudo Operations

The pseudo instruction CON (constant) can be used to include data in a program (for example, object prologues):

CON(n) $q_1 \ldots q_n$ $q_n \ldots q_1$

Exercises

10-1. Assemble the following program (it does not perform any particular function—its purpose is to be an exercise in assembly):

```
       CON(5)    #02DCC
begin  CON(5)    (end)-(begin)

       GOTO      l1

sub1   A=A-1     A
       LCHEX     #12345
l2     C=C-1     A
       GONC      l2
       RTNCC

l1     LCHEX     #00005
       A=C       A
l3     GOSUB     l2
       ?A#0      A
       GOYES     l3

       LCHEX     #00001
       A=C       A
       GOXUB     l4
       ?A=0      A
       GOYES     l5
       A=A-1     A
l5     A=DAT0    A
       D0=D0+    5
       PC=(A)

l4     ?C=0      A
       RTNYES
       C=0       A
       A=A+1     A
       RTN
end
```

10-2. Using the table in the appendix, disassemble the following code:

```
14313 31791 577B7 61557 13114 21648 08C
```

11. HP 48 Objects

The HP 48 handles things called objects. There are 28 of them, 2 of which are indirectly accessible to the user (indicated by one star), and 13 of which are not accessible at all in the standard manner (indicated by two stars). These objects always begin with a 5-nibble prolog number that indicates their nature. Following is a list of all the objects with their prolog number and their type (returned by the function TYPE):

Prolog	Object		Type
02911	System Binary	(**)	20
02933	Real		0
02955	Long Real	(**)	21
02977	Complex		1
0299D	Long Complex	(**)	22
029BF	Character	(**)	24
029E8	Array		3/4
02A0A	Linked Array	(**)	23
02A2C	String		2
02A4E	Binary Integer		10
02A74	List		5
02A96	Directory		15
02AB8	Algebraic		9
02ADA	Unit		13
02AFC	Tagged		12
02B1E	Graphic		11
02B40	Library	(**)	16
02B62	Backup	(*)	17
02B88	Library Data	(**)	26
02BAA	System Binary	(**)	27
02BCC	System Binary	(**)	27
02BEE	System Binary	(**)	27
02C10	System Binary	(**)	27
02D9D	Program		8
02DCC	Code	(**)	25
02E48	Global Name		6
02E6D	Local Name	(*)	7
02E92	XLIB Name	(**)	14

Each of these 28 objects possesses a well-defined structure that we will study in detail. Each object will be presented in table form with explanations for each element of the table.

As you read this chapter, keep in mind that the microprocessor reverses the values that it reads. This means that values are written backwards to memory, including the prologs given here. Thus, the prolog 02911 would be written 11920 in the HP 48's memory.

Note that all values in memory are stored in hexadecimal, regardless of the current binary base mode (binary, octal, decimal, or hexadecimal).

System Binary Object

@	Prolog (02911)	5 nibbles
@+5h	Content	5 nibbles
@+Ah		

A system binary is a short binary integer (5 nibbles) that is used internally by the HP 48. It appears on the screen in the form <*XXXXXb*> where *XXXXX* is the contents and *b* is the current binary base. In particular, it can be used to pass parameters between two different programs.

Examples

- `1192000000` is the system binary <00000h>;
- `1192054321` is the system binary <12345h>;

Exercises

11-1. What does `1192012345` represent?

11-2. Code the system binary <ABCDEh>;

11-3. Code the system binary <123d>.

Real Number Object

@	Prolog (02933)	5 nibbles
@+5h	Exponent	3 nibbles
@+8h	Mantissa	12 nibbles
@+14h	Sign	1 nibble
@+15h		

This is the usual real number accessible by the user. The code is separated into 3 parts: The sign, the mantissa (a number from 1 to 9, inclusive), and the exponent. Together these form the real number:

$$\text{sign} * \text{mantissa} * 10^{\text{exponent}}$$

The three parts are coded internally in the following manner:

- If the exponent is negative, it is replaced by "1000 - exponent" in order to obtain a positive number. This number from 0 to 999 is stored in Binary Coded Decimal using 3 nibbles. This is why the HP 48 can have exponents from -499 to +499.
- The mantissa is multiplied by 10^{11} to make it an integer, and it is stored in Binary Coded Decimal using 12 nibbles.
- The sign is coded in 1 nibble, using 0 for positive and 9 for negative.

Examples

- `12345.6789` is coded as `3392040000098765432810`.
- `-3.14159265359E-2` is coded as `33920899953562951413 9`.

Exercises

11-4. Code the real number `12`.

11-5. What does `339204000000000543779` represent?

Long Real Number Object

@	Prolog (02955)	5 nibbles
@+5h	Exponent	5 nibbles
@+Ah	Mantissa	15 nibbles
@+19h	Sign	1 nibble
@+1Ah		

This object is used internally by the HP 48 for calculations needing more precision. The coding principle is the same as the real number, except that the exponent can have a value in the range [-49999,49999], and the mantissa can have 15 significant digits.

Examples

- 55920000009798535629514130 represents the long-real approximation of π: 3.1415926535897
- The long real -123E45678 would be represented by 55920876540000000000003219

Exercises

11-6. How would the HP 48 code the long real 1234567890123456?

11-7. What does 55920899990000000000000019 represent?

Complex Number Object

@	Prolog (02977)		5 nibbles
@+5h	Exponent 1	*real part*	3 nibbles
@+8h	Mantissa 1		12 nibbles
@+14h	Sign 1		1 nibble
@+15h	Exponent 2	*imaginary part*	3 nibbles
@+18h	Mantissa 2		12 nibbles
@+24h	Sign 2		1 nibble
@+25h			

The structure of a complex number is simple. After the 5-nibble prolog, there are two real numbers without prologs, the first being the real part of the complex number, and the second being the imaginary part.

Example

- The complex number (123456789012, 210987654321) is coded 7792011021098765432101101234567890120

Exercises

11-8. Code the complex number (1, 2).

11-9. What does the following code represent?
7792010000000000003391000000000000330

Long Complex Number Object

Address	Field	Part	Size
@	Prolog (0299D)		5 nibbles
@+5h	Exponent 1	*real part*	5 nibbles
@+Ah	Mantissa 1		15 nibbles
@+19h	Sign 1		1 nibble
@+1Ah	Exponent 2	*imaginary part*	5 nibbles
@+1Fh	Mantissa 2		15 nibbles
@+2Eh	Sign 2		1 nibble
@+2Fh			

The long complex is similar to the complex number, with the two real numbers being long reals.

Example

- The long complex (123456789012345,543210987654321) is coded:
  `D9920110005432109876543210110001234567890123450`

Exercises

11-10. Code the long complex (0,0).

11-11. What does this represent?
`D9920000005432109876543219110001234567890123459`

Character Object

@	Prolog (029BF)	5 nibbles
@+5h	Character	2 nibbles
@+7h		

This is simply a number from 0 to 255 (00h to FFh), which represents a character. The extended ASCII character codes can be found in the HP 48 manuals.

Example

- `FB92014` is the character A (A is ASCII code 41h).

Exercises

11-12. Code the character C (ASCII code 43h).

11-13. What does `FB92044` represent?

Real/Complex Array Object

@	Prolog (029E8)	5 nibbles
@+5h	Total length excluding prolog l_t	5 nibbles
@+Ah	Type of objects (of length l_o)	5 nibbles
@+Fh	Number, d, of dimensions	5 nibbles
@+14h	Dimension 1 (d_1)	5 nibbles
	⋮	
@+d*5+14h	Dimension d (d_d)	5 nibbles
@+d*5+19h	Contents of object 1	l_o nibbles
	⋮	
	Contents of object d_2+1	l_o nibbles
	⋮	
@+l_t-l_o+5h	Contents of object d_1*...*d_d	l_o nibbles
@+l_t+5h		

The array object is used for storing vectors and matrices. In fact, there is no difference between a vector and a matrix.

Just after the length of the object is given the object type of the array contents. This type number (5 nibbles long) is actually the prolog number of the objects. For this reason an array can only contain objects of the same type. Notice also that the dimension is not restricted to 1 (vector) or 2 (matrix). This number can be just about as large as you like.

Next come the dimension sizes. For a matrix, this would be the number of rows and columns.

After this come the actual values stored in the array object. These values are objects themselves without a prolog (which is not necessary since it was given earlier in the declaration part of the array). These objects are arranged in order of dimensions. For example, a two-dimensional matrix would be stored as row 1, then as row 2 since the first dimension of a matrix is its number of rows.

It must be noted that although it is possible to create matrices with many dimensions (like a 25 dimensional matrix containing vectors), they are not very useful because the HP 48 does not handle them correctly.

Example

- The matrix `[[1 2][3 4]]` is coded as:
  `8E920 95000 33920 20000 20000 20000`
  `000000000000001 0 000000000000002 0`
  `000000000000003 0 000000000000004 0`

Exercises

11-14. Give the first 35 nibbles of a 3x5x8 matrix containing system binary numbers.

11-15. What type of elements are contained in a matrix that begins with the following code?
`8E92010F00C2A20100009100052000` ...

Linked Array Object

@	Prolog (02A0A)	5 nibbles
@+5h	Total length excluding prolog l_t	5 nibbles
@+Ah	Type of objects (of length l_o)	5 nibbles
@+Fh	Number, d, of dimensions	5 nibbles
@+14h	Dimension 1 (d_1)	5 nibbles
@+d*5+14h	Dimension d (d_d)	5 nibbles
@+d*5+19h	Pointer to object 1	5 nibbles
	Pointer to object d_2+1	5 nibbles
	Pointer to object d_1*...*d_d	5 nibbles
	Element 1	l_o nibbles
@+l_t-l_o+5h	Element n	l_o nibbles
@+l_t+5h		

Linked arrays are arrays where the elements have been replaced by pointers to objects found at the end of the array. A NULL pointer indicates the absence of an element.

This structure permits a more economical storage for matrices that have many identical elements. In the following example the identity matrix of order 2 can be stored in 82 nibbles instead of 94.

Example

- This is the code for the identity matrix of order 2:

```
A0A20 D4000 33920 20000 20000 20000 41000 F1000
A1000 50000 0000000000000010 0000000000000000
```

String Object

@	Prolog (02A2C)	5 nibbles
@+5h	Total length excluding prolog l_t	5 nibbles
@+Ah	First character	2 nibbles
	⋮	
@+l_t-2h	Last character	2 nibbles
@+l_t+5h		

The coding of a string is simple. It consists of a prolog, followed by the total length of the string, followed by a list of ASCII character codes.

Example

- `"STRING"` is coded as: `C2A20 11000 35 45 25 94 E4 74`

Exercises

11-16. Code the string `"Hello World"`.

11-17. Decode this object: `C2A203100024271667F60212`

Binary Integer Object

@	Prolog (02A4E)	5 nibbles
@+5h	Total length excluding prolog I_t	5 nibbles
@+Ah	Binary integer value	I_t-5 nibbles
@+I_t+5h		

The maximum length of a binary integer is normally 15h (this is the length of a 16 digit hexadecimal binary integer), but you can increase this length considerably. In fact, the HP 48 uses large binary integers internally.

Example

- #12345678h is coded as E4A2051000876543210000000O

Exercises

11-18. Code the binary integer #87654321d

11-19. Decode E4A20A00012345

List Object

Address	Field	Size
@	Prolog (02A74)	5 nibbles
@+5h	First object	
	⋮	
	Last object	
	Epilog (0312B)	5 nibbles

A list is simply a list of objects. Its structure consists of a prolog, a list of objects, then an epilogue. You can think of the prolog as the list delimiter `{` and the epilogue as the list delimiter `}`.

Example

- `{"A" B}` is coded as `47A20C2A20700001484E201024B2130`

Exercises

11-20. Code an empty list.

11-21. Decode `47A2084E2020F4B4B2130`

Directory Object

@	Prolog (02A96)	5 nibbles
@+5h	Number of attached libraries, n_1	3 nibbles
@+8h	N° Library	3 nibbles
@+Bh	Address of Hash Table 1	5 nibbles
@+10h	Address of Message Table 1	5 nibbles
	Library n_1	3 nibbles
	Address of Hash Table n_1	5 nibbles
	Address of Message Table n_1	5 nibbles
$@_1$	Offset to last object ($@_d$-$@_1$)	5 nibbles
$@_1$+5h	00000	5 nibbles
$@_2$	n_1, characters in $name_1$	2 nibbles
$@_2$+2h	Character 1, $name_1$ — *name of object 1*	2 nibbles
	character n_1	2 nibbles
	n_1 characters in $name_1$	2 nibbles
	Object 1	
$@_3$	Size of previous fields ($@_3$-$@_2$)	5 nibbles
$@_3$+2h	n_2 characters in $name_2$	2 nibbles
$@_d$	n_d characters in $name_d$	2 nibbles
$@_d$+2h	Character 1, $name_d$ — *name of object d*	2 nibbles
	Character n_d	2 nibbles
	n_d characters in $name_d$	2 nibbles
	Object d	

There are two different types of directories. The first type is the HOME directory, which is the root directory of the VAR menu. Any number of libraries may be attached to this directory. The second type is a subdirectory, found either in the HOME directory, or one of its subdirectories. We will first look at the structure of the HOME directory, shown in the table above.

Notice that in the code for a directory you will find information about any libraries that might be attached. The first field after the prolog indicates the number of libraries attached.

Next comes a series of descriptor fields for each attached library . This field is divided into three parts:

- The library number: This number is assigned according to the following criterion defined by Hewlett-Packard:
  - #000h to #100h HP lib. in ROM;
  - #101h to #200h HP lib. in RAM;
  - #201h to #300h non HP lib. (distributed by HP);
  - #301h to #6FFh free use;
  - #700h to #7FFh used internally by the HP 48.
- The address of the hash table for the library (see page 143).
- The address of the message table of the library (see page 143). This pointer is NULL if there is no message table.

Note:

- The HOME directory always has a minimum of 2 libraries attached to it: library #002h and library #700h.
- If the address pointers are pointing to tables in the hidden ROM (see **Chapter 12**), then an indirect address is given. The address points to a system binary in normal ROM which contains the address of the object in the hidden ROM.

The beginning of a subdirectory is different than the HOME directory:

@	Prolog (02A96)	5 nibbles
@+5h	Number of the attached library	3 nibbles
@+8h	Offset to last object ($@_d$-$@_1$)	5 nibbles
@+Dh	00000	5 nibbles
@+12h	n_1 characters in $name_1$	2 nibbles
	⋮	

If there is no attached library, then #7FFh will appear in the library number field. The rest of the code is the same for both kinds of directories. The next field contains an offset to the last object in the directory. Immediately following this field is 5 zero nibbles to mark the first object in the directory. This is useful when searching the directory backwards.

Each variable contained in the directory is defined with the following fields:

- The number of characters in the name (in 2 nibbles);
- The characters of the name (in ASCII code);
- The number of characters in the name (in 2 nibbles);
- The object;
- The total length of the 4 fields just mentioned—useful when searching the directory backwards (the last object in the directory does not have this field).

Examples

- This is the code for an empty directory: `69A20FF700000`
- A directory that contains a `3` in a variable named `'D'`:
  `69A20FF7A0000000001044 10C2A207000033`

Exercises

11-22. Add the variable `'A'`, containing `4`, to the directory in the example above.

11-23. Attach library #123h with a hash table found at address #7FE30h and without a message table to the directory above.

Algebraic Object

@	Prolog (02AB8)	5 nibbles
@+5h	First object	
	⋮	
	Last object	
	Epilog (0312B)	5 nibbles

The algebraic expression represented by this object is stored in RPL form. For this reason, there is no need to store parenthesis.

The operations are coded by their address in ROM (in 5 nibbles). This address points to the code that executes the desired algebraic function.

Example

- `'C+D'` is coded in the form `C D +` by:
  `8BA2084E20103484E20104476BA1B2130`

Exercises

11-24. Code the expression `'A+B'`.

11-25. The subtraction routine is found at address #1AD09h and the multiplication routine is found at address #1ADEEh. Knowing that, decode the following object:
`8BA2084E20101484E20102484E20103490DA1EEDA1B2130`

Unit Object

@	Prolog (02ADA)	5 nibbles
@+5h	Object implied	
	Desc 1 ⋮ Desc n — *unit description*	
	Epilog (0312B)	5 nibbles

After the prolog comes the object implied by the unit. This is actually part of an RPL calculation that describes its relation to the unit. The elementary units themselves are stored in the form of object strings.

Only 3 operations are possible between units—all related to multiplication (because it is not possible to create a unit by adding joules to seconds or by subtracting grams from kilometers):

• Multiplication • Division • Raise to a power

Each operation is represented by a reference number to an object found in ROM. The following table is useful in coding or decoding unit objects:

Operation	*	/	^
Reference	#10B86h	#10B68h	#10B72h

Example

- 9.81_m/s^2 is coded as
  ADA2033920000000000000001890C2A2070000D6 ...
  ...C2A20700003733920000000000000002027B01 ...
  ...86B0168B01B2130 (Actually, the HP 48 would replace the real number 2 by a pointer to a real number found in ROM).

Exercises

11-26. Code the following: 1.2_m.

11-27. Decode: ADA20339200000000000000150C2A2070000D63 ...
...F2A227B0168B01B2130

Tagged Object

@	Prolog (02AFC)		5 nibbles
@+5h	Length l_t of the tag		2 nibbles
@+7h	Character 1	*characters of the tag*	2 nibbles
	⋮		
@+l_t*2+5h	Character l_t		2 nibbles
@+l_t*2+7h	Object		

This object has a prolog, the number of characters in the tag, the characters themselves (in ASCII), and then the tagged object.

Example

- `REAL:1.23456789012` is coded as:
  `CFA2040255414C4339200002109876543210`

Exercises

11-28. Code `UN:TAG`

11-29. Decode `CFA2020F4B484E206034F4252514C4`

Graphics Object

@	Prolog (02B1E)		5 nibbles
@+5h	Total length excluding prolog l_t		5 nibbles
@+Ah	Number n_l of lines (in pixels)		5 nibbles
@+Fh	Number n_c of columns (in pixels)		5 nibbles
@+14h	Columns 1 to 8	*Pixels in line 1*	1+1 nibbles
	⋮		
	Last pixels		1+1 nibbles
	⋮		
	Columns 1 to 8	*Pixels in line n_l*	1+1 nibbles
	⋮		
	Last pixels		1+1 nibbles
@+l_t+5h			

The dimensions of a graphics object are always given in pixels and stored with a number of columns that is divisible by 8. Zero columns are added if the number of columns is not already divisible by 8.

The first nibble stores the first 4 columns; the next nibble stores the next 4 columns, etc. The least significant bit of these nibbles is the left-most column, and the most significant bit is the right-most column.

Example

- `GROB 8 1 FF` is coded as: `E1B201100010000800000FF`

Exercise

11-30. Decode: `E1B2011000100004000000F0`

Library Object

@	Prolog (02B40)	5 nibbles
@+5h	Total length excluding prolog l_t	5 nibbles
@+Ah	n_c characters in name	2 nibbles
@+Ch	Character 1	(2 nibbles)
	Characters of the name	
$@+n_c$*2 +Ah	Character n_c	(2 nibbles)
$@+n_c$*2 +Ch	n_c characters in name	(2 nibbles)
$@+n_c$*2 +Eh	Library number	3 nibbles
$@_1$	Offset to Hash Table ($@_h$-$@_1$)	5 nibbles
$@_2$	Offset to Message Table ($@_m$-$@_2$)	5 nibbles
$@_3$	Offset to Link Table ($@_l$-$@_3$)	5 nibbles
$@_4$	Offset to Config. Object ($@_c$-$@_4$)	5 nibbles
$@_h$	Hash Table	
$@_m$	Message Table	
$@_l$	Link Table	
$@_{o1}$-(7,9)h	Type $XLIB_1$ (command/function)	1 or 3 nibbles
$@_{o1}$-6h	Library number of $XLIB_1$	3 nibbles
$@_{o1}$-3h	Command number of $XLIB_1$	3 nibbles
$@_{o1}$	Object $XLIB_1$	
	⋮	
$@_{on}$-(7,9)h	Type $XLIB_n$ (command/function)	1 or 3 nibbles
$@_{on}$-6h	Library number of $XLIB_n$	3 nibbles
$@_{on}$-3h	Command number of $XLIB_n$	3 nibbles
$@_{on}$	Object $XLIB_n$	
$@_{o(n+1)}$	Other object 1	
	Other objects (not visible)	
$@_{o(n+m)}$	Other object m	
$@_c$	Config. Object (not visible)	
$@+l_t$+1h	Checksum (CRC)	4 nibbles
$@+l_t$+5h		

The library is the most complex of all HP 48 objects. The code begins with the optional library name (in an unnamed library, the fields for the name characters and the second field for the name length are absent). After the name comes the library number, which must be unique (see **Directory**

Object). Next are 4 offsets —to the hash table, message table, link table and configuration object (executed after each system halt). A NULL field means that a table or the object does not exist. After the offsets come the 3 tables, in any order, if they exist. After the tables come the library's visible objects, each preceded by its command number (3 nibbles before), its library number (6 nibbles before), and a flag coded in either 1 or 3 nibbles:

- If it is a library of commands (library number ≥ #700h), the flag will be only 1 nibble. Its significance is not clear, but the value 9h (1001b) seems best. The command itself is composed of 2 objects: first, the object used when the command is executed; second, the object used during the coding phase of the command line.

- If it is a library of functions (library number ≤ #6FFh), then *if bit 3 of the nibble at* $@_{on}$*-7h is 0*, the function can be included in an algebraic object, and the flag is 3 nibbles long. The bits mean the following:

Nibble at $@_{on}$*-8h*:	bit 0	Unknown 12 (12)	bit 1	Unknown 11 (11)
	bit 2	INT (10)	bit 3	RULES (9)
Nibble at $@_{on}$*-9h*:	bit 0	Unknown 8 (8)	bit 1	Unknown 7 (7)
	bit 2	ISOL (6)	bit 3	DER (5)
Nibble at $@_{on}$*-7h*:	bit 0	ALG (4)	bit 1	Unknown 3 (3)
	bit 2	EQWR (2)	bit 3	0 (alg. obj. OK)

  Each bit signifies the presence or absence of a special program for the function (ISOL to invert, DER for derivative, INT for integration, RULES to add functions in a sub-menu, ALG for algebraics; EQWR for the EquationWriter). The function itself is a series of objects, led by the program for the function. The others are supplemental functions in the order of the numbers in parentheses. For example, if the flag value is #C81h, there will be a principal program, PRG, plus ALG, DER, RULES and INT, in that order. The code: <C81> *<Lib number> <Xlib number>* <PRG> <ALG> <DER> <RULES> <INT>

  If bit 3 of the nibble at $@_{on}$*-7h is 1*, then it is a command (just like those in libraries with numbers > #700h). The flag is coded in 3 bits. The other bits are different than for the regular library commands (bit 1 seems to indicate that the command also exists in function form).

  The library checksum is calculated for the zone from @+5h to $@+l_t+1h$ according to the formula described with the backup object.

To minimize library access time, the HP 48 uses *hashing*: A function takes the name of a command and returns a number from #1h to #10h (the HP 48 uses the number of characters in the name). For each class, a part of the table then gives the addresses of the name and number of each command in that class. Here is the hash table structure:

$@_h$	Prolog (02A4E)	5 nibbles
$@_h$+5h	Total length excluding prolog l_h	5 nibbles
$@_{c1}$	Offset for class 1 ($@_{n1}$-$@_{c1}$)	5 nibbles
	⋮	
$@_{c16}$	Offset for class 16 ($@_{n16}$-$@_{c16}$)	5 nibbles
@+5Ah	Length l_n of the name list	5 nibbles
$@_{n1}$	Number of characters in name	2 nibbles
$@_{n1}$+2h	First character — *Characters in name 1*	2 nibbles
	⋮	
	Last character	2 nibbles
	Command number 1	3 nibbles
	⋮	
$@_{nx}$	Number of characters in name x	2 nibbles
$@_{nx}$+2h	First character — *Characters in name x*	2 nibbles
	⋮	
	Last character	2 nibbles
	Command number x	3 nibbles
	⋮	
@+l_n+5Ah	Offset to cmd name 1 ($@_{ox}$-$@_{n1}$)	5 nibbles
	⋮	
$@_{ox}$	Offset to cmd name x ($@_{ox}$-$@_{nx}$)	
	⋮	
	offset to the last command name	
$@_h$+l_h+5h		

The hash table is one large binary integer. The first 16 fields are offsets to the starts of each name table. The next field contains the length of the entire name table. The name table is a list of these elements (in this order): The name length, the name characters (in ASCII), the command number. The last field gives (by command number) the offsets used to find the command names in the table—used to display the names in the menu bar .

The message table has the following structure:

$@_m$	Prolog (029E8)		5 nibbles
$@_m$+5h	Total length excluding prolog l_m		5 nibbles
$@_m$+Ah	Object types: string (02A2C)		5 nibbles
$@_m$+Fh	Number of dimensions (00001)		5 nibbles
$@_m$+14h	n number of messages		5 nibbles
$@_m$+19h	Length l_1		5 nibbles
$@_m$+1Dh	First character	*Text for message 1*	2 nibbles
	⋮		
$@_m+l_1$+19h	Last character		2 nibbles
	⋮		
	Length l_n		5 nibbles
	First character	*Text for message n*	2 nibbles
	⋮		
	Last character		2 nibbles
$@_m+l_m$+5h			

This is a vector that contains strings (for more information on vectors, see **Real/Complex Array Object**). This vector contains messages that are used by the library. The message number corresponds to its place in the vector. The internal library #002h uses such a table to store the HP 48's error messages.

The link table has the following structure:

$@_l$	Prolog (02A4E)	5 nibbles
$@_l$+5h	Total length excluding prolog l_l	5 nibbles
$@_{l1}$	Offset to object 1 ($@_{o1}$-$@_{l1}$)	5 nibbles
	⋮	
$@_{ld}$	Offset to object d ($@_{od}$-$@_{ld}$)	5 nibbles
$@_l$+l_l+5h		

The link table is used for finding the address of the beginning of a library object. The link table is really just a large binary integer containing a series of 5 nibble offsets. These offsets are in the same order as the library objects.

Example

- An empty library is coded as
  04B20C0004006594445440FF60000000000000000000049B1

Exercises

11-31. What is the library number of the above example?

11-32. What is the library name?

11-33. Does this library have a message table?

Backup Object

@	Prolog (02B62)		5 nibbles
@+5h	Total length excluding prolog l_t		5 nibbles
@+Ah	n_c number of characters		2 nibbles
@+Ch	Character 1	*Object name*	2 nibbles
	⋮		
@+n_c*2 +8h	Character n_c		2 nibbles
@+n_c*2 +Ah	n_c number of characters		2 nibbles
@+n_c*2 +Ch	First Backup object		
	⋮		
	Last Backup object		
@+l_t+5h			

This is the object used for storing backups in a port. After the prolog and the length fields is a field with the backup object' s name, followed by each object being backed up.

Normally, a backup object contains two objects: the object being backed up and a system binary containing the CRC (Cyclic Redundancy Code, or checksum) of the object. This type of backup object structure is shown below:

@	Prolog (02B62)		5 nibbles
@+5h	Total length excluding prolog l_t		5 nibbles
@+Ah	n_c number of characters		2 nibbles
@+Ch	Character 1	*Text for message 1*	2 nibbles
	⋮		
@+n_c*2 +8h	Character n_c		2 nibbles
@+n_c*2 +Ah	n_c number of characters		2 nibbles
@+n_c*2 +Ch	Object		
@+l_t-5h	Prolog 02911	*System Binary containing CRC*	5 nibbles
@+l_t	0		1 nibbles
@+l_t+1h	CRC value		4 nibbles
@+l_t+5h			

A backup object contains only one object, followed by a system binary, which contains the checksum of the object. This sum is calculated using the same formula used to calculate the CRC in a library. The formula used is also the same control code used by the Kermit protocol for data transmission, that is, the remainder of a division by the polynomial:

$$x^{16}+x^{12}+x^{5}+1$$

The HP 48 does not perform this calculation with software. Rather, it is a hard-wired function performed by a specialized circuit (see **Chapter 13**). The CRC program presented in the **Library of Programs** does the same calculation using software. For a backup object, this checksum is calculated over the area from @+5h to @+l_t, inclusive.

Example

- `26B209200040243 4B40540C2A2090000F4B41192006D26`
  is the code for the backup object containing the string: `"OK"`.

Exercises

11-34. What is the name of the backup object in the above example?

11-35. What is its checksum?

Library Data Object

@	Prolog (02B88)	5 nibbles
@+5h	Total length excluding prolog l_t	5 nibbles
@+Ah	Contents	l_t-5 nibbles
@+l_t+5h		

This object does not exist as a basic object for the HP 48. It can be used only in a library for storing data of any type. It could be used, for example, in a mini-spreadsheet library needing to store spreadsheets in a form different than that used for matrices.

There is no standard structure for this object except that it begins with its prolog (as does every object), followed by its length, then data.

Reserved 1, 2, 3 and 4

@	Prolog	5 nibbles
@+5h	Total length excluding prolog l_t	5 nibbles
@+Ah	Contents	l_t-5 nibbles
@+l_t+5h		

These four objects have the same structure as the library data object. They are not used, and are probably reserved for a future use. In this way , Hewlett-Packard can create a new object without needing to completely re-structure the existing ROM.

The prologs are:

- #02BAAh for Reserved 1;
- #02BCCh for Reserved 2;
- #02BEEh for Reserved 3;
- #02C10h for Reserved 4.

Since these objects don't actually exist, no examples or exercises will be given here.

Program Object

@	Prolog (02D9D)	5 nibbles
@+5h	First object	
	⋮	
	Last object	
	Epilog (0312B)	5 nibbles

This object is used to store all user programs. Its structure is similar to that of a list: a prolog, a collection of objects (of any type), and an epilogue. However, the prolog and epilogue do not correspond to the « and » program delimiters, as these are objects that must be included in the list.

Example

- The program `« A B + »` is coded as:
  `D9D20E163284E20101484E20102476BA193632B2130`

Exercises

Refer to the above example to answer these questions:

11-36. How are the program delimiters, « and », coded?

11-37. How is the addition function (+) coded?

Code Object

@	Prolog (02DCC)	5 nibbles
@+5h	Total length excluding prolog l_t	5 nibbles
@+Ah	Machine code	l_t-5 nibbles
@+l_t+5h		

This object is used to store machine language programs. The "machine code" field contains a series of machine language instructions.

Example

- See the machine language programs in the **Library of Programs** for examples.

Exercises

11-38. How would you code an empty code object?

11-39. Using what you have learned from other chapters, write some machine language code that does nothing.

Global Name Object

@	Prolog (02E48)		5 nibbles
@+5h	n_c number of characters		2 nibbles
@+7h	Character 1	*Characters of the name*	2 nibbles
	⋮		
$@+n_c*2$ +3h	Character n_c		2 nibbles
$@+n_c*2$ +5h			

This object is used for storing global names. The field following the prolog indicates the number of characters in the name, followed by the characters themselves (in ASCII).

Example

- The global name `'Journey'` is coded as:
  `84E2070A4F65727E65697`

Exercises

11-40. Code `'Hello'`.

11-41. What does `84E2000` represent?

Local Name Object

@	Prolog (02E6D)		5 nibbles
@+5h	n_c number of characters		2 nibbles
@+7h	Character 1	Characters of the name	2 nibbles
	⋮		
@+n_c*2 +3h	Character n_c		2 nibbles
@+n_c*2 +5h			

This object is used to store local variable names. Its structure is the same as the global name (above) except for the prolog.

Example

- 'Local' is coded as: D6E2050C4F63616C6

Exercises

11-42. How many characters are in this local name?
D6E2040E416D656

11-43. What is that name?

XLIB Name Object

@	Prolog (02E92)	5 nibbles
@+5h	Library number	3 nibbles
@+8h	Command number	3 nibbles
@+Bh		

The XLIB name is a method used to reference library commands. In order to optimize access to these commands, their name is replaced by an "XLIB name" which contains the library number and the command number of the command in question. This notation can by used to access the two standard HP 48 libraries (library #002h and library #700h).

Example

- The FREE command, which is library #002h, command number #163h, can be represented as: 29E20200361

Exercises

11-44. Code command number #123h from library #456h.

11-45. What are the library and command numbers of the XLIB name: 29E20100200?

Other Objects

Any of the objects found in ROM may be added to your own objects. For example, if you wanted to add a few RPL commands to your machine language program, it is easy, using the method below. In fact, if you have need of an RPL command, a common list, a machine language command, or any other object found in ROM, here is how you could add one of these to your object:

- RPL commands, lists, and other composite objects (listed in the **Appendix**) can be added using their address only. For example, the SWAP instruction can be represented by the ROM address #1FBBDh.
- Machine language routines stored in the form < *current address* + 5h> <*machine code*>, or, more commonly, <*address of an ML program*>. This method can be used only with objects in ROM where their address is fixed. These objects are shown on the screen as <External>, or, in other words, an external call.

12. General Memory Organization

We have previously seen that the Saturn microprocessor has 20-bit address registers and can thus address as many as 2^{20} memory elements. Since these basic memory elements are nibbles, the HP 48 can address 1 "Mega-nibble," which is 512 Kb (Kilobytes). This memory space is divided into 5 parts:

- ROM: This contains all programs used by the machine (square roots, curve tracing, beep, etc.). This memory can not be modified, and has a size of 256 Kb.

- I/O RAM: This 64-nibble memory area is used to access the HP 48 peripherals (infrared receiver/transmitter, clock, screen, etc.). The I/O RAM is actually part of the ROM memory area.

- Built-in RAM: This is where all user data is stored (programs, variables, alarms, etc.). The size of this memory area is 32 Kb.

- Plug-in card ports (2): Each of the ports can contain 1 card of up to 128 Kb.

Notice, however, that if you total the maximum amount of possible memory (with two 128 Kb cards installed), the result is 544 Kb, which is 32 Kb larger than what the Saturn microprocessor is capable of addressing.

To overcome this problem, the HP 48 uses a technique called *bank-switching*. Bank-switching assigns two distinct memory areas to the same address, with one having priority over the other. This higher-priority memory is visible; the other is "hidden." If you want to access the hidden memory, you must reconfigure the visible memory, to give it another address. The hidden memory area is then accessible.

In order to minimize access time, the only thing that should be stored in the hidden memory area is data that is infrequently used. The HP 48 stores the auto-test routines, error messages, etc.).

The HP 48 memory is therefore in one of two states:

- The standard state, where the built-in RAM occupies the memory area from #70000h to #7FFFFh (see **Figure 1** opposite).
- An information access state where the built-in RAM is displaced to address #F0000h. The HP 48 is in this state when using the mini-editor (see **Figure 2**).

The mini-editor permits easy access to this second memory state, and thus allows access to all the memory contents of the calculator . To use this mini-editor, enter the manual auto-test (by pressing [ON]-[D]), then press the [←] key. This editor uses one line of the screen to display 16 nibbles of memory at the current address. The following commands may be used:

- [0], [1], [2],...[9], [A],...[F] changes the value at the current address (to be used with caution!);
- Movement commands:
  - By #1000h with [▼] and [▲]
  - By #100h with [×] and [÷]
  - By #1h with [+] and [−]
- Serial port output commands:
  - By #10h with [·]
  - By #10000h with [SPC]
- Commands for accessing pre-defined memory areas:
  - #00100h (I/O RAM) by [ENTER]
  - #80000h (Port 1) by [EEX]
  - #C0000h (Port 2) by [DEL]
  - #F000Ah (WSLOG data) by [+/−]
  - #F0A8Ch (screen area) by [1/x]
- To update the screen: [←];
- To execute the machine language program beginning at the current address: [EVAL] (to be used with caution!).

For the HP 48SX, when viewing the plug-in card contents, these contents appear at memory locations #80000h and #C0000h, although they are reconfigured to form a continuous memory area when used normally by the machine.

Address	Contents		Size
#00000h	Beginning of ROM		256 nibbles
#00100h	I/O RAM		64 nibbles
#00140h	Continuation of ROM		458432 nibbles
#70000h	Built-in RAM		65536 nibbles
#80000h	Port 1	*Plug-in*	262144 nibbles
#C0000h	Port 2	*cards*	262144 nibbles
#100000h			

Figure 1: HP 48 memory, standard state

Address	Contents		Size
#00000h	Beginning of ROM		256 nibbles
#00100h	I/O RAM		64 nibbles
#00140h	Continuation of ROM		523968 nibbles
#80000h	Port 1	*Plug-in*	262144 nibbles
#C0000h	Port 2 (partial)	*cards*	196608 nibbles
#F0000h	Built-in RAM (displaced)		65536 nibbles
#100000h			

Figure 2: HP 48 memory, information access state

13. I/O RAM

To communicate with its peripherals, the HP 48 uses, among other methods, a special memory area called the I/O RAM. This 64 nibble area is a way to exchange data with the outside world. By reading and writing to this area, it is possible to send commands or receive data from the peripherals.

In the following pages, the I/O RAM will be described bit by bit using tables in the form shown below. In these tables, bit 3 is the nibble's most significant bit, and bit 0 is the least significant.

	Bit 3	Bit 2	Bit 1	Bit 0
#00100h				
#00101h				

	Bit 3	Bit 2	Bit 1	Bit 0
#00100h	Display	Left margin		
#00101h	Screen contrast			
#00102h				
#00103h				
#00104h	CRC calculator			
#00105h				
#00106h				
#00107h				
#00108h	Batt. test			
#00109h				
#0010Ah				
#0010Bh	Alert	Alpha	right shift	left shift
#0010Ch	annunciator		transmitting	Busy
#0010Dh			RS232 speed	
#0010Eh				
#0010Fh	Port information (HP 48SX)			
#00110h	RS 232C interrupts			
#00111h				
#00112h			input OK	output OK
#00113h				
#00114h	RS 232C Input			
#00115h				
#00116h	RS 232C Output			
#00117h				
#00118h				
#00119h				
#0011Ah	IR input			IR in mem
#0011Bh				
#0011Ch	IR output			
#0011Dh				
#0011Eh				
#0011Fh	Base address of built-in RAM			

Left Margin

The left margin is coded with 3 bits and therefore may have a value from 0 to 7. It can be used for scrolling the main screen portion (everything but the menu bar). For example, setting the left margin to 1 shifts the screen contents one pixel to the left. To use the left margin properly, you will need to understand the right margin and the address of the screen bitmap, both of which are described later.

Display

Setting display to 0 turns off the screen display; setting it to 1 reactivates it. Interestingly, turning off the screen deactivates the keyboard, and accelerates the machine by about 13%. This is because the screen bitmap is in memory: if the screen is off, there is no memory access each time the screen is updated. With this small burden lifted from the bus, exchanges between the microprocessor and memory can be done more quickly, and so program execution will be faster. The program FAST (see the **Library of Programs**) uses this method to achieve rapid calculations.

Screen Contrast

The screen contrast is coded with 5 bits (the most significant bit being at #00102h). Therefore, the contrast can be adjusted to 32 levels. However, only the values from #3h to #13h are accessible by pressing [ON]-[+] and [ON]-[−]. The program CONTRAST (see the **Library of Programs**) uses this address to adjust the contrast from software.

CRC Calculator

The HP 48 uses checksums to verify the integrity of data (see **Chapter 4**). In order to obtain this value rapidly, a hardware circuit is used for the calculation. This circuit reads the information going between the micro-processor and memory and calculates the corresponding CRC (Cyclic Redundancy Code).

To calculate the CRC of an object (just as the function BYTES does), set the four nibbles to zero (nibbles #00104h to #00107h), then read the nibbles of the object in question. The CRC of that object will then be found in nibbles #00104h to #00107h.

This process must not be interrupted, so you must disable interrupts while the calculation is taking place (using the assembly instruction `INTOFF`). Don't forget to re-enable interrupts when the calculation is finished (using the assembly instruction `INTON`).

Because these four nibbles are constantly changing, they are very useful for generating random numbers in a machine language program. As the CRC value is a function of nibbles read from memory, you can read a pseudo-random number (for example, the clock, the address of the stack end, the amount of free memory, etc.), then read the pseudo-random number contained at #00104h.

Battery Test

The nibbles #00108h and #00109h are used for testing the HP 48's batteries (main batteries as well as batteries for the plug-in cards in the case of the HP 48SX).

To begin the test, set bit 3 of nibble #00109h to 1 (by writing #Ch, the other 3 bits being 1, 0, and 0, respectively). Then, read the contents of nibble #00108h. Each of the bits of this nibble indicates the state of one of the batteries of the HP 48:

- If bit 3 of #00108h is 1, the plug-in card battery for port 2 is weak;
- If bit 2 of #00108h is 1, the plug-in card battery for port 1 is weak;
- If bit 1 of #00108h is 1, the HP 48's main batteries are weak;
- If bit 0 of #00108h is 1, the main batteries are very weak.

Note that the HP 48's internal battery tester reads the nibble #00108h many times (6). If one of these reads returns a 1, then the battery is declared weak.

When you finish the testing, don't forget to change bit 3 of #00109h back to 0 (by writing a #4h to #00109h).

Annunciators

The annunciators (α, ⧗, etc.) each have 2 states controlled by one bit (1=showing, 0=not showing). Bit 3 of #0010Ch determines whether any of the annunciators will be showing (0=none showing, 1=showing, according to their respective states).

RS-232C Speed

The transmission and reception of data from the RS-232C port is done at a speed expressed as a "baud" rate. This number refers to the number of bits transmitted per second.

The HP 48 is capable of transferring data at four different speeds: 1200 baud, 2400 baud, 4800 baud, and 9600 baud. Bits 1 and 2 of #0010Dh are used to set this speed, as follows:

Bit 2	Bit 1	RS-232C Speed
0	0	1200 Baud
0	1	2400 Baud
1	0	4800 Baud
1	1	9600 Baud

Port Information (HP 48SX)

Nibble #0010Fh gives the states of the two ports for the HP 48SX. The possible states are:

Bit Number		Significance
0	When set (1):	Card present in port 1
1	When set (1):	Card present in port 2
2	When set (1):	Card in port 1 not write-protected
3	When set (1):	Card in port 2 not write-protected

RS-232C Interrupts

When a character is sent to the HP via the RS-232C port, this can cause an interrupt. This would cause the microprocessor to execute a special interrupt handling routine. For example, if a character is received through the RS-232C port, then the character needs to be read and then stored in the RS-232C buffer (see **Chapter 14**).

The nibble #00110h can be used to disable these interrupts as well as determine if one has occurred. Each bit of this nibble has a distinct function:

Bit Number	Significance
0	When set (1): a character was received; an interrupt has occurred.
1	When set (1): receive interrupts are enabled.
2	When set (1): a character was transmitted; an interrupt has occurred.
3	When set (1): transmission interrupts are enabled.

To access the RS-232C port directly, you should disable these interrupts.

Input OK and Output OK

If the Input OK bit is set, then a character has just been received via the RS-232C port. You may read this value from nibble #00114h.

If the Output OK bit is set, then you may output a character to the RS-232C port by writing to #00116h.

RS-232C Input and Output

Input and output through the RS-232C port are accomplished by a special circuit. To receive a byte from this port, read the two nibbles at #001 14h.

To transmit a byte through the RS-232C port, write the two nibbles at #00116h.

IR Input and Output

Nibble #0011Ah is used for IR input. Bit 3 is set if there was a reception; it is clear if there was not. Bit 0 is set at the first reception and serves as a reminder that there was an IR input. This bit must be set back to 0 manually.

Bit 3 of nibble #0011Ch is used for IR output. Setting this bit to 1 begins the transfer, 0 stops it.

Base Address of Built-in RAM

#0011Fh contains the base address of the built-in RAM (#7h or #Fh). #7h is the normal value (built-in RAM is at #70000h); #Fh means that the built-in RAM has been displaced to #F0000h. This value is brought up to date by the system when the reconfiguration takes place (in order to view the hidden ROM).

Changing the value in #0011Fh has no effect on the base address of the built-in RAM; it is for reading only. This nibble is used by routines that must function in normal mode, as well as when the RAM is displaced (like the routine that updates the screen). In this way, the location of the built-in RAM makes no difference, and the machine is still capable of functioning.

	Bit 3	Bit 2	Bit 1	Bit 0
#00120h	Beginning address of screen bitmap			
#00121h				
#00122h				
#00123h				
#00124h				
#00125h	Right margin (in nibbles)			
#00126h				
#00127h				
#00128h	Menu bar height & VSYNC			
#00129h				
#0012Ah				
#0012Bh				
#0012Ch				
#0012Dh				
#0012Eh				
#0012Fh				
#00130h	Beginning address of menu bar bitmap			
#00131h				
#00132h				
#00133h				
#00134h				
#00135h				
#00136h				
#00137h	Timer 1			
#00138h	Timer 2			
#00139h				
#0013Ah				
#0013Bh				
#0013Ch				
#0013Dh				
#0013Eh				
#0013Fh				

Screen Bitmap Address

The HP 48 screen is divided into the screen itself (where the stack appears) and the menu bar (at the bottom). The information for these portions may be stored at any address, but the screen driver must know that address. The bitmap for the main screen is pointed to by #00120h. The memory at that location is simply a GROB containing the screen contents.

- This address must be even (because a specialized circuit is used that manages 8-bit screen portions only).
- This address can only be written to, but a readable duplicate of this address is located in the reserved RAM (see **Chapter 14**).

Right Margin

The right margin for the screen bitmap is stored at #00125h. This value is defined in nibbles, not in pixels as is the left margin. This number must be even, so bit 0 is ignored. To perform rapid screen scrolling, change the left and right margins and the address pointing to the beginning of the bitmap, and the screen will display the new area of the bitmap. The value contained at #00125h follows the same rules as the bitmap address: It cannot be read, but its value is backed up in the reserved RAM area.

Menu Bar Height

The separation height between the main screen area and the menu bar is defined in #00128h. Setting this value to #3Fh causes the menu bar to disappear. The value at this location cannot be read, so it is backed up in the reserved RAM area. The standard values (with no library attached):

- #7097Ch for the screen bitmap address (stack GROB);
- #70858h for the menu bitmap address;
- #000h for the right margin; #0h for the left margin;
- #37h for the separation height.

VSYNC

We have seen that the menu bar height can only be written to. This is because the nibbles #00128h and #00129h are also used for the VSYNC. If you read the contents of these nibbles, you will get the line number that the screen driver is currently working on during a screen refresh. This will be a number that goes from #3Fh down to #0h every 1/64th of a second.

Timer 1

The nibble at #00137h is a 1/16th -second timer that counts down from #Fh to #0h every second.

Clock

The last area in the I/O RAM is for the clock. Its value is in units of 1/8192 seconds, and is stored in an 8 nibble area, decreasing from #FFFFFFFFh to #00000000h. The HP 48 does not actually use this entire value.

- If the clock is visible on the screen, the machine counts down in one-second cycles. Every second, the value of these 8 nibbles goes from #00001FFFh to #00000000h (or 8192 8192^{nds} of a second).
- If the clock is not visible on the screen, and if an alarm is due in the next hour, then the number of 8192^{nds} remaining until the alarm is stored in the clock section.
- If neither of the above is true, then the values used are from 0 to 1 hour (or #01C20000h to #00000000h) returning to 1 hour when a button is pressed in interactive mode.

Each time the clock value reaches #00000000h an interrupt is generated.

14. RAM

The HP 48 memory is divided into several zones, each with a distinct role. Before getting into the details of each zone, here is a representation of the entire memory:

#70000h	Reserved RAM	
(#70551h)	Screen GROBS	
(#7056Ah)	Temporary objects	
(#7056Fh)	Return stack	
B	Free memory	D*5 nibbles
D1	The stack	(#7069Fh) nib.
(#7057Eh)	Command line	48 nibbles min.
(#70583h)	Undo stack, local variables	
(#70588h)	5 zeros	5 nibbles
(#7058Dh)	Temporary environment	78 nibbles
(#70592h)	User variables (HOME dir)	
(#70597h)	Backup in port 0	
(#70669h)		

All of these zones, except the reserved RAM, are at variable addresses. These addresses are stored in the reserved RAM (and certain registers). We will describe the reserved RAM, and its contents in detail.

#70000h	CMOS word		5 nibbles
#70005h	0000		4 nibbles
#70009h	Disable system-halt		1 nibble
#7000Ah	Type	*WSLOG 1*	1 nibble
#7000Bh	Date		13 nibbles
#70018h	CRC		4 nibbles
#7001Ch	Type	*WSLOG 2*	1 nibble
#7001Dh	Date		13 nibbles
#7002Ah	CRC		4 nibbles
#7002Eh	Type	*WSLOG 3*	1 nibble
#7002Fh	Date		13 nibbles
#7003Ch	CRC		4 nibbles
#70040h	Type	*WSLOG 4*	1 nibble
#70041h	Date		13 nibbles
#7004Eh	CRC		4 nibbles
#70052h	Value	*Clock offset*	13 nibbles
#7005Fh	CRC		4 nibbles
#70063h	0000000000000		13 nibbles
#70070h	FF		2 nibbles
#70072h	Auto-test start time		13 nibbles
#7007Fh	Auto-test fail time		13 nibbles
#7008Ch	Mini editor screen preparation		44 nibbles

CMOS Word

The 5 first nibbles in reserved RAM are always #A5C3Fh, used to verify the reserved RAM contents. Changing these values causes a system halt.

Disable System Halt

Setting bit 3 of nibble #70009h will disable the system halt ON-C, manual auto-test ON-D, and automatic ON-E. It also makes it impossible to turn the machine off; it is automatically turned back on after a moment.

WSLOG

Data about the WSLOG command is stored in nibbles #7000Ah, #7001Ch, #7002Eh, and #70040h. This command, (not documented in the HP manuals), returns the cause and time of the machine's last warm boot. The cause is coded (from #0h to #Fh) in the first nibble of the zone:

Code	Cause of Warm Boot
0	The machine was turned on while in the COMA mode (COMA mode is entered by pressing ON-SPC).
1	Batteries are very weak.
2	A hardware problem occurred during an infrared transmission.
3	The machine experienced a restart (execution of the program at #00000h).
4	The clock offset (controlled by CRC) was corrupted.
5	An uncontrolled data change occurred in one of the plug-in cards.
6	Not used.
7	A verification word (5 nibbles) in RAM does not correspond to the memory state (RAM is probably corrupted).
8	An error was detected while configuring one of the 5 peripherals. One of them is not configured, or the configuration does not correspond to a valid peripheral.
9	The alarm list is corrupted (its CRC is not valid).
A	Not used.
B	Plug-in card removed.
C	System reset (using the reset button found underneath one of the machine's rubber feet).
D	RPL error manager not found.
E	Configuration table corrupted.
F	RAM card removed.

Next is the date of the warm boot (in 8192 nds of a second since January 1, 0001), coded in 13 nibbles. The final 4 nibbles are a checksum for the 14 preceding nibbles, calculated as in **Chapter 11** (and as in CRC in the **Library of Programs**).

Clock Offset

At #70052h is found the clock of fset (13 nibbles), followed by its checksum (4 nibbles). As before, this offset is in units of 1/8192 seconds beginning at January 1, 0001.

Autotest Start & Fail Time

The two 13 nibble zones at #70072h and #7007Fh are used during the auto-test to store the test starting time, and the fail time respectively (if a fail occurs). As these values have little importance, they are not validated with a CRC.

Mini-Editor Screen Preparation

The 44 nibbles at #7008Ch are for preparing the display during the use of the mini-editor (22 characters).

Address	Contents		Size
#700B8h	???...???		35 nibbles
#700DBh	Plug-in cards (bits 0 and 1)		1 nibble
#700DCh			288 nibbles
#701FCh	Data	*Input buffer for the RS 232C port*	512 nibbles
#703FCh	BufLen		2 nibbles
#703FEh	BufFull		1 nibble
#703FFh	BufStart		2 nibbles
#70401h			39 nibbles
#70428h	CRC for the configuration table		4 nibbles
#7042Ch	Flags	*Information for the plug-in card in port 1*	1 nibble
#7042Dh	Size		5 nibbles
#70432h	Start		5 nibbles
#70437h	Flags	*Information for the plug-in card in port 2*	1 nibble
#70438h	Size		5 nibbles
#7043Dh	Start		5 nibbles
#70442h			11 nibbles
#7044Dh	End of Built-in RAM backup zone		5 nibbles
#70452h	End of port 1 backup zone		5 nibbles
#70457h	End of port 2 backup zone		5 nibbles
#7045Ch	Temporary backup during interrupts		103 nibbles
#704C3h	Output mask for keyboard test		3 nibbles
#704C6h			16 nibbles

Plug-In Cards (HP 48SX)

This nibble, #700DB, is the same as in the I/O RAM at address #0010Fh:

Bit 3	Bit 2	Bit 1	Bit 0
1 = Port 2 not write-protected	1 = Port 1 not write-protected	1= Plug-in card present in Port 2	1= Plug-in card present in Port 1

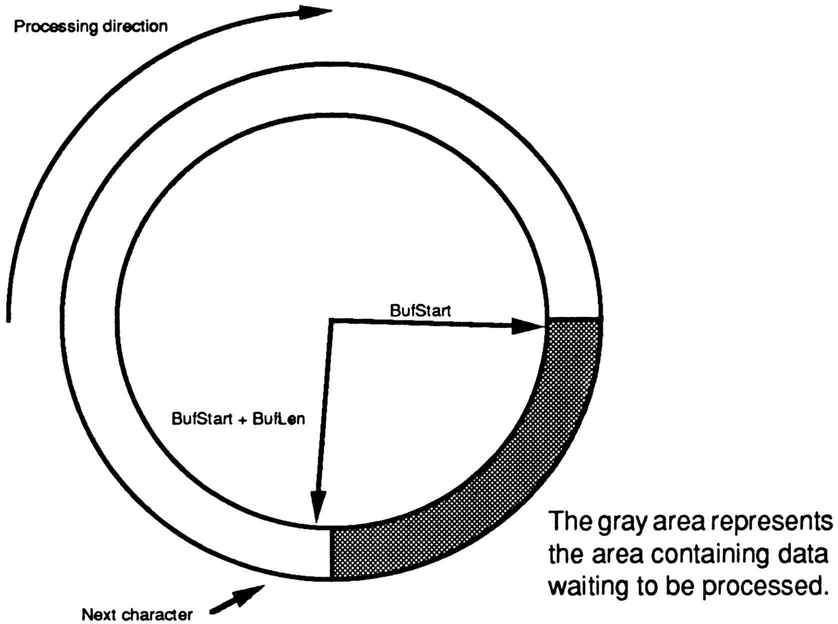
For example, if nibble #700DBh contains #Bh (#101 1b), this means that: a plug-in card is in port 1 (bit 0 set); a plug-in card is in port 2 (bit 1 set); port 1 is write-protected (bit 2 clear); port 2 is not write-protected (bit 3 set).

RS-232C Input Buffer

The RS-232C input buffer temporarily stores data coming from the exterior still needing to be processed. It consists of:

- A data block of 512 nibbles (256 characters) that begins at #701FCh;
- A starting pointer, *BufStart* (2 nibbles at #703FFh), the number of the first character in the buffer. Its address is #701FCh+2*BufStart.
- A character counter, *BufLen* (2 nibbles at #703FCh). The address of the last character received is #701FCh+2*BufStart+2*BufLen-2. The next character will be stored at #701FCh+2*BufStart+2*BufLen;
- A full indicator, *BufFull* (1 nibble at #703FEh) which is used to indicate if the buffer is full. This nibble is 0 if the buffer is not full, 8 if information was lost.

The buffer can be represented by this diagram:



Configuration Table

The 37 nibbles beginning at #70428h are a configuration table describing the state of the plug-in cards. The first 4 nibbles of this table are a checksum for the other 33 nibbles. This checksum is not calculated by the usual CRC formula, but by a machine-language routine at #09B73h, which returns the checksum in field **A** of register **C**.

Plug-In Card Information (HP 48SX)

These two 11 nibble blocks are part of the configuration table. Nibble #7042Ch contains information for the plug-in card in port 1 (#70437h for port 2).

This first nibble in the block consists of the following information:

- Bit 1 is set if the card is merged with RAM;
- Bit 2 is set if the card is not write-protected.;
- Bit 3 is set if the card is present

The next 5 nibbles (beginning at #70432h and #7043Dh) contain the starting address of the plug-in card. And the size of the card (0's complement) is stored at #7042Dh and #70438h. A 32 Kb card will have a value of #F0000h; a 128 Kb card will have a value of #C0000h. These values (the starting address and size) are not valid if the card is merged with RAM.

The next 11 nibbles (at #70442h) are also part of the configuration table and are probably reserved for future use.

Backup End

The three groups of 5 nibbles found at #7044Dh, #70452h and #70457h contain, respectively: the ending addresses of the backup zones for the built-in RAM, the card in port 1, and the card in port 2. Note that if a card is merged with built-in RAM, its backup zone is also merged.

To calculate the free space of a plug-in card that is not merged, simply use the configuration table and the three addresses mentioned above. The program BFREE in the **Library of Programs** uses this technique, which allows it to calculate the free space even if the card is write-protected (this is not possible using the function PVARS).

Caution: ROM cards (which look like write-protected RAM cards to the HP 48SX) may return false values if the data are not stored on the card using the "normal" card BACKUP techniques. In particular, these data can be found in memory after the theoretical end of the card.

Interrupt Backup

The 103 nibble block at #7045Ch is used by the system during interrupts to temporarily backup the register contents. Interrupts are used by the HP 48 for processing keypresses, the RS-232C port, the clock, etc.

Output Mask for the Keyboard Test

The output mask at #704C3h is used as an argument for OUT=C for a keyboard test done by an interrupt handling routine. It is set to #1FFh by the system. Periodically setting these 3 nibbles to #FFFh will cause the speaker to sputter since interrupts occur every second.

Machine Speed

The 5 nibbles at #704D6h contain the machine speed in number of cycles per sixteenths of a second. To obtain the microprocessor speed, multiply this value by 16. The following program calculates the machine speed using the programs PEEK and STR→A found in the **Library of Programs**.

```
SPD (# 4BC5h)
  « # 704D6h #5 PEEK STR→A
    16 * B→R 1_Hz →UNIT
  »
```

Invert the result to find the duration of one clock cycle—useful for calculating the execution time of a machine-language program (see **Chapter 10**). If you change these 5 nibbles to a larger value, all sounds will have a higher pitch (but this does not mean that the processor has been accelerated).

Disable Keyboard

Nibble #704DCh is used to disable the keyboard. Setting this nibble to a non-zero value will accomplish this (#Fh for example). Note:

- Neither the [ON] button nor the system halts are disabled.
- Disabling the keyboard does not disable interrupts associated with pressing certain buttons, but simply disables the execution of the normal keyboard processing routine (the key codes will not be stored in the keyboard buffer).
- This nibble is set to zero by the system when the calculator returns to interactive mode (at the end of program execution, for example).

Key State

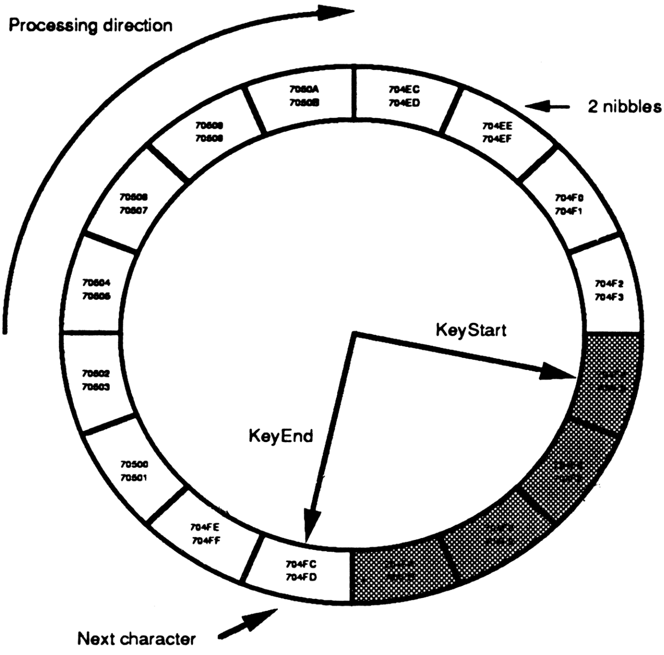
This 13 nibble block, beginning at #704DDh, stores the current state of the HP 48's 49 buttons. One bit per button is set if the button is being pressed. This table is updated each time a keypress interrupt occurs.

Keyboard Buffer

The keyboard buffer is a 32-nibble block beginning at #704ECh. Each key code is 2 nibbles long, so this buffer can hold 16 key codes. The buffer contains only key presses that have not yet been processed. Two pointers are used to keep track of the buffer contents:

- *KeyStart* indicates the position number of the first button pressed.
- *KeyEnd* indicates the first free position number (where the next key code will be stored).

The yet-to-be-processed key codes are therefore contained in nibbles #704E2h+2*KeyStart to #704EC+2*KeyEnd. This is a circular buffer similar to the RS-232C buffer:



(In this diagram, KeyStart equals 4 and KeyEnd equals 8)

A	B	C	D	E	F
01	02	03	04	05	06
MTH	PRG	CST	VAR	▲	NXT
07	08	09	0A	0B	0C
'	STO	EVAL	◀	▼	▶
0D	0E	0F	10	11	12
SIN	COS	TAN	$\sqrt{x}$	y^x	$1/x$
13	14	15	16	17	18
ENTER	+/−	EEX	DEL	⬅	
19	1A	1B	1C	1D	
α	7	8	9	÷	
80	1F	20	21	22	
↰	4	5	6	×	
40	24	25	26	27	
↱	1	2	3	−	
C0	29	2A	2B	2C	
ON	0	•	SPC	+	
2D	2E	2F	30	31	

Key codes stored in the keyboard buffer

#704D6h	Machine speed	5 nibbles
#704DBh		1 nibble
#704DCh	Disable keyboard	1 nibble
#704DDh	Key state	13 nibbles
#704EAh	KeyStart *Keyboard*	1 nibble
#704EBh	KeyEnd *buffer*	1 nibbles
#704ECh	Key codes	32 nibbles
#7050Ch		2 nibbles
#7050Eh	Screen bitmap addr. (#00120h)	5 nibbles
#70513h	Right margin (#00125h)	3 nibbles
#70516h	Menu bitmap address (#00130h)	5 nibbles
#7051Bh	Menu height (#00128h)	2 nibbles
#7051Dh		52 nibbles
#70551h	@ of menu GROB	5 nibbles
#70556h	@ of stack GROB	5 nibbles
#7055Bh	@ of current GROB	5 nibbles
#70560h	@ of PICT GROB	5 nibbles
#70565h	@ of PICT GROB ?	5 nibbles
#7056Ah	Beginning @ of temporary objects	5 nibbles
#7056Fh	Ending @ of temporary objects	5 nibbles
#70574h	Beginning @ of free mem. (B)	5 nibbles
#70579h	Ending @ of free memory (D1)	5 nibbles

Backups

In **Chapter 13** we saw that several blocks of ROM were used to define the HP 48's display (left margin, right margin, menu height, etc.), but some of these could not be read. For this reason, they have been stored in the reserved RAM area.

- The address of the screen bitmap is stored at #7050Eh (#00120h).
- The right margin is stored at #70513h (#00125h).
- The address of the menu bitmap is stored at #70516h (#00130h).
- The separation height between the main screen section and the menu bar is stored at #7051Bh (#00128h).

These parameters are always stored in two locations (reserved RAM, and I/O RAM) by the HP 48 screen management routines.

Graphics Object Addresses

The following 5 addresses point to different graphics objects used by the machine:

- #70551h stores the address of the menu bar GROB.
- #70556h stores the address of the stack GROB.
- #7055Bh stores the address of the current GROB (stack or PICT).
- #70560h stores the address of the PICT GROB.
- #70565h also stores the address of the PICT GROB.

These objects are all stored in the temporary object memory area.

Temporary Objects

#7056Ah and #7056Fh are beginning and ending addresses that define a memory area used for storing temporary objects. This area is for objects that won't last long or that change frequently, such as stack objects, intermediate results used by the machine, display preparation, etc. Each of these objects is stored with the following format:

Flag (garbage collector)	1 nibble
Object	l_z - 6 nibbles
Object length l_z	5 nibbles

As you use the machine, these objects accumulate in the temporary object memory area. It is necessary to do a clean-up from time to time to purge the temporary objects that are no longer being used. This clean up procedure (which is called each time the command MEM is executed) is done by a program called the "garbage collector." This program can be called with a GOSBVL to address #0613Eh.

During this operation, the machine marks (in the flag area of the structure shown above) each of the temporary objects that are still being used. After having checked each object, the HP 48 purges the objects that are not marked. The temporary memory area has the following structure:

(#7056Ah)	00000	5 nibbles
	Flag	1 nibble
	Object	
	Length	5 nibbles
	⋮	
	Flag	1 nibble
	Object	
	Length	5 nibbles
(#7056Fh)		

Return Stack

The ending address of the temporary object memory area is also the beginning address of the return stack. If a program is called within a program, this stack stores the return address to the original program. An address is placed on the stack when the program prolog is encountered (#02D9Dh), and an address is taken from the stack when an epilog is encountered (#031B2h), which indicates the end of a program.

Register B points to the end of this memory area (which is generally backed up at #70574h). Here is a representation of the return stack:

(#7056Fh)	Return address 1	5 nibbles
	Return address 2	5 nibbles
	⋮	
	Return address n	5 nibbles
(B)		

In this list, address 1 is the oldest. Register **B** points to the end of this stack, which is the beginning of free memory. Since the routine SAVE_REG (#0679Bh) saves **B** at #70574h, the value of **B** is often found there.

Free Memory

The free memory is the area between the address contained in **B** (end of return stack) and the address contained in **D1** (which points to the first level of the stack). The size of the free memory is stored in register **D** (field **A**) as the number of 5-nibble "blocks" that are free. For example, if field **A** of **D** was #00100h, this would indicate that the amount of free memory is between #00500h and #00504h nibbles.

The "blocks" are 5 nibbles because the return stack and the user stack also use blocks of 5 nibbles each. This makes it easy to know if there is enough free memory to extend one of these stacks, (which is a frequent operation): all the machine has to do is check to see that field **A** of **D** is non-zero.

The User Stack

Just as **B** is backed up in #70574h, register **D1**, the stack pointer, is backed up in #70579h. The HP 48 stack may contain any object. Internally, the stack contains only addresses that point to objects, because addresses all have the same size: 5 nibbles. Register D1 points to the first level of the stack. The stack ends at the location pointed to by #7057Eh:

(D1)	Address of object in level 1	5 nibbles
	Address of object in level 2	5 nibbles
	⋮	
	Address of the last object	5 nibbles
	00000	5 nibbles
(#7057Eh)		

To find the address of an object in level n, simply take the value of **D1**, add $(n-1)*5$, and read the 5 nibbles at that address. The following assembly program duplicates the SWAP function:

```
A=DAT1 A      * Address of object 1
D1=D1+ 5      * Now pointing to level 2
C=DAT1 A      * Address of object 2
DAT1=A A      * Write address of object 1
D1=D1- 5      * Now pointing to level 1
DAT1=C A      * Write address of object 2
```

Caution: This program does not check the size of the stack.

The Command Line

The command line begins at the address stored in #7057Eh and ends at the address stored in #70583h. This memory area contains the command line that is currently being edited.

The command line consists of ASCII character codes terminated by the null character, which serves as an end of line delimiter. This explains why you can't edit strings containing the null character. The command line is always at least 23 characters in length, plus the null character. Nonexistent characters are replaced by "nulls."

(#7057Eh)	Character 1	2 nibbles
	Character 2	2 nibbles
	⋮	
	Character n ($n \geq 23$)	2 nibbles
	00	2 nibbles
(#70583h)		

The Undo Stack

A copy of the stack contents (the Undo stack) and local variables are stored in the same memory area. This area is divided into blocks:

(#70583h)	Block 1	
	Block 2	
	⋮	
	Last block (undo)	
	00000	5 nibbles
(#70588h)		

The last block is the copy of the stack contents (UNDO); the others are local variables and their contents—from most recent to oldest. Each of these blocks is divided into several fields:

@	Total length L	5 nibbles
	Block identifier	5 nibbles
	Address of the first local name	5 nibbles
	Address of the first contents	5 nibbles
	⋮	
	Address of the last local name	5 nibbles
	Address of the last contents	5 nibbles
@+L		

For local variables, the block identifier is #00000h. A local name address points to an object of the form '*local name*'. The address of the contents points to the object stored in the local variable of the name preceding it.

For the undo stack, the structure is similar. The block identifier is #00001h if there are no local variables; #00002h otherwise. To remain consistent with the local variable block structures, we find pointers to local names in the undo stack block structure—all pointing to the same address, #61D3Ah, which is an address (in ROM) of the empty local name ('').

@	Total length L	5 nibbles
	Block identifier	5 nibbles
	Address of '' (#61D3Ah)	5 nibbles
	Number of elements on the stack	5 nibbles
	Address of '' (#61D3Ah)	5 nibbles
	Address of the object in level 1	5 nibbles
	⋮	
	Address of '' (#61D3Ah)	5 nibbles
	Address of the object in level n	5 nibbles
@+L		

The other fields contain the addresses of the objects in the undo stack and the depth of the stack.

Temporary Environment

The temporary environment is used for managing the menus. This memory area contains the necessary addresses for displaying the menu labels and for executing the associated routines.

The display addresses help the HP 48 determine the text to be displayed in the menu label, as well as the text to place in the command line in PRG or ALG modes. The execution addresses are used to find the address of the program associated with a menu item. If a menu label has no associated function, its name is the empty name (address #055DFh) and the execution address is #3FDD1h, which is a program that makes a "beep."

It seems that a block has been reserved for a seventh menu item. This could be for future use, or, perhaps when these structures were first made, the menu size was not completely decided.

(#7058Dh)	#07Ch	3 nibbles
(#7058Dh)+3h	Address of menu label 1	5 nibbles
(#7058Dh)+8h	Address of menu label 2	5 nibbles
(#7058Dh)+Dh	Address of menu label 3	5 nibbles
(#7058Dh)+12h	Address of menu label 4	5 nibbles
(#7058Dh)+17h	Address of menu label 5	5 nibbles
(#7058Dh)+1Ch	Address of menu label 6	5 nibbles
(#7058Dh)+21h	Address of menu label 7 (reserved)	5 nibbles
(#7058Dh)+26h	Execution address 1	5 nibbles
(#7058Dh)+2Bh	Execution address 2	5 nibbles
(#7058Dh)+30h	Execution address 3	5 nibbles
(#7058Dh)+35h	Execution address 4	5 nibbles
(#7058Dh)+3Ah	Execution address 5	5 nibbles
(#7058Dh)+3Fh	Execution address 6	5 nibbles
(#7058Dh)+44h	Execution address 7 (reserved)	5 nibbles
(#7058Dh)+49h		

Home Directory

At #70592h is a pointer to a directory object containing the home directory . This directory is entered after a system halt, or the execution of the command HOME. This object is described in detail in **Chapter 11**. This address is stored again at #705A1h.

Current Directory

The address of the current directory, which is also a directory object, is stored at #7059Ch.

Backup Area

The HP 48 is capable of making backups, either for a plug-in card (for the HP 48SX) or for the built-in RAM (in port 0).

The backup area is organized in the same manner regardless of the port used. In the case of the built-in RAM, (or that of the built-in RAM merged with a plug-in card for the HP 48SX), we find the address of the beginning of this area at #70597h. This memory area consists of a list of backup objects (see **Chapter 11**).

Backup object 1	
Backup object 2	
Last backup object	
00000	5 nibbles

#705A6h	@ of user key assignments	5 nibbles
#705ABh	@ of alarm list	5 nibbles
#705B0h	Pointer to next object to be evaluated	5 nibbles
#705B5h	Backup area	5 nibbles
#705BAh	LAST object 1	5 nibbles
#705BFh	LAST object 2 *LAST*	5 nibbles
#705C4h	LAST object 3	5 nibbles
#705C9h	LAST object 4 *Stack*	5 nibbles
#705CEh	LAST object 5	5 nibbles
#705D3h	Large binary (table for internal use?)	5 nibbles
#705D8h	00000	5 nibbles
#705DDh	Command 1 *Stack of the*	5 nibbles
#705E2h	Command 2 *four most recent*	5 nibbles
#705E7h	Command 3 *command lines*	5 nibbles
#705ECh	Command 4	5 nibbles
#705F1h	?????	5 nibbles
#705F6h	?????	5 nibbles
#705FBh	?????	5 nibbles
#70600h	@ of last error message	5 nibbles
#70605h		25 nibbles
#7061Eh	Current menu	5 nibbles
#70623h	Last menu	5 nibbles
#70628h		15 nibbles
#70637h	Unshifted menu key routine	5 nibbles
#7063Ch	Left-shifted menu key routine	5 nibbles
#70641h	Right-shifted menu key routine	5 nibbles
#70646h	Review key	5 nibbles
#7064Bh		20 nibbles
#7065Fh	Last RPL token	5 nibbles
#70664h		5 nibbles
#70669h	@ of the End of RAM	5 nibbles
#7066Eh	Free memory (5 nibble blocks) (**D**)	5 nibbles

User Keys and Alarms

At #705A6h and #705ABh are the addresses of the user key assignments and the alarm list, respectively. The two tables found at these addresses are actually variables like any other user-created variables, except they are stored in a hidden directory.

It is actually possible to "hide" objects stored in the user directory. The principle is simple: If, during a clean-up of the current directory (done periodically to determine the names of the objects in this directory), the machine comes across an object with the empty name (''), it stops its search. To hide an object, you could either give it the name '' (which is what the HP 48 does for the directory that contains the user key assignments and alarm list), or you could store it after an object with the empty name. In this case, the object is executable but its name doesn't appear in a menu label.

The HP 48's hidden directory contains the following objects:

- 'Alarms' contains the alarm list;
- 'UserKeys' contains the definition list for user-key assignments;
- 'UserKeys.CRC' contains the checksum for UserKeys (calculated via UserKeys BYTES DROP).

To access this hidden directory, simply go to the home directory and type #15781h SYSEVAL. You then find yourself in the hidden directory (the SYSEVAL simply evaluates the empty name, '').

Access to different hidden objects is also possible, but be advised never to purge or even modify them, lest you experience Memory Lost. To return to the home directory, just type HOME.

Next Object to be Executed

#705B0h serves as a backup for the register **D0** and therefore points to the next object to be executed.

LAST Stack

The LAST stack is a list of five addresses that point to objects being temporarily saved (so the maximum number of objects saved by LAST ARG is 5 even though only three parameters will usually be saved). If fewer than 5 objects are being saved, the other addresses are set to #00000h.

Address of a Large Binary Integer

At #705D3h is the address of a large binary integer (184 digits). It is probably a table used internally by the HP 48. This object is stored in the temporary environment. Since it is the first temporary object created by the HP 48, it is always the first object found in this part of RAM.

Command Line Stack

The command line stack is based on the same principle as the LAST stack. It consists of four addresses pointing to character strings that contain the last four command lines. The address of the most recent command line is contained in #705DDh; the oldest is in #705ECh.

Address of Last Error Message

At #70600h is the address of a character string which contains the last error message, if it was an error defined by the user (via "*message*" DOERR). Otherwise, this address is set to #00000h.

Menus

At #7061Eh and #70623h are the addresses of the current menu, and the last menu, respectively. The menu offsets are stored at #707C9h (current menu) and #707CEh (last menu). The menus, or the objects pointed to by these addresses, are lists. The content of these lists is identical to that of the custom menu (CST) defined by the user (see **Chapter 5**).

An element of these menu lists may be one of the following:

- *A name*: The name is placed in the menu label and is considered to be the name of an executable object. Just like in the VAR menu, if you press the menu button itself, then the object of that name is executed. If you first press the left shift, then the object in level one of the stack is stored under the menu name. If you first press the right shift, then the contents of the object are recalled to the stack.
- *A character string*: The contents of the string serve as a name to be placed in the menu label, and if the button is pressed, then the contents of the string are added to the command line.
- *A 21x8 GROB*: This GROB will be used for the menu label.
- *A list*:
  - The first element of the list will be used as the menu label. If this element is a program object (prolog D9D20) that first contains the address #40788h, this object will be executed, and its result will be used as a menu label (string, GROB, etc.). Any program object beginning with `D9D2088704` will be executed. Four addresses are particularly useful:

    #3A328h takes a string from the stack and returns the corresponding graphics object as it would appear in the menu label.

    #3A3ECh takes a string from the stack and returns a subdirectory label GROB.

    #3A44Eh takes a string from the stack and returns an inverse menu label GROB (like in the SOLVR menu).

    #3A38Ah takes a string and returns a menu label GROB such as in the MODES menu (with a white box beside the name).

Note that since these particular program objects are executed when the menu label is displayed, you can use this concept in the CST menu to display special messages immediately after entering the menu (just like the TIME menu, for example).

- The second element of the list determines the action taken when the menu button is pressed. It can also be a list whose first element corresponds to the action taken when the menu button is pressed by itself, the second element is if the left shift was pressed first (↰), and the third element is if the right shift was pressed first (↱).

No.	Menu	Address	No.	Menu	Address
0	Last Menu		30	SOLVE.SOLVR	#15200h
1	CST	#3B239h	31	PLOT	#3BEB8h
2	VAR	#3F6D8h	32	PLOT.TYPE	#3C039h
3	MTH	#3B284h	33	PLOT.PLOTR	#3C0AFh
4	MTH.PARTS	#3B36Ch	34	ALGEBRA	#3C483h
5	MTH.PROB	#3B3E4h	35	TIME	#3C4C9h
6	MTH.HYP	#3B420h	36	TIME.ADJST	#3C671h
7	MTH.MATR	#3B452h	37	TIME.SET	#3C79Ch
8	MTH.VECTR	#3B489h	38	TIME.ALRM	#3C8D5h
9	MTH.BASE	#3B4CAh	39	TIME2	#3C9B8h
10	PRG	#3B542h	40	STAT	#3CAA7h
11	PRG.STK	#3B622h	41	STAT.MODL	#3CD96h
12	PRG.OBJ	#3B67Fh	42	UNITS	#3CE65h
13	PRG.DSPL	#3B6F7h	43	UNITS.LENG	#3D08Ch
14	PRG.CTRL	#3B7E2h	44	UNITS.AREA	#3D1F3h
15	PRG.BRCH	#3B8B4h	45	UNITS.VOL	#3D2D6h
16	PRG.TEST	#3B90Eh	46	UNITS.TIME	#3D451h
17	PRINT	#3B972h	47	UNITS.SPEED	#3D4BAh
18	I/O	#3B9A4h	48	UNITS.MASS	#3D553h
19	I/O.SETUP	#3BA03h	49	UNITS.FORCE	#3D642h
20	MODES	#3BB46h	50	UNITS.ENRG	#3D6B5h
21	MODES2	#3BC8Dh	51	UNITS.POWR	#3D764h
22	MEMORY	#3BCE7h	52	UNITS.PRESS	#3D797h
23	MEMORY2	#3BD46h	53	UNITS.TEMP	#3D838h
24	LIBRARY	#3F376h	54	UNITS.ELEC	#3D887h
25	PORT0	#3BD82h	55	UNITS.ANGL	#3D93Ah
26	PORT1	#3BDAAh	56	UNITS.LIGHT	#3D9B3h
27	PORT2	#3BDD2h	57	UNITS.RAD	#3DA42h
28	EDIT	#3BDFAh	58	UNITS.VISC	#3DABFh
29	SOLVE	#3BE22h	59	UNITS2	#3DAF2h

Last RPL Token

At #7065Fh is the address of the object that caused the command line to be executed. If the ENTER key caused the execution, then the address corresponds to an empty program object. If a VAR menu button was pressed to cause the execution, then the address of the name of the object to be executed will be stored here.

The End of RAM

The address of the end of RAM is stored at #70669h. The HP 48SX RAM can be extended by adding one or more plug-in RAM cards. As each card is added, the memory is reconfigured such that the user memory forms one contiguous block. The program RAMSEARCH in the **Library of Programs** uses this address to determine the memory area to search.

Free Memory

The five nibbles at #7066Eh are used to backup register **D**, which contains an approximation of the free memory. The value given is the number of 5-nibble blocks that are available. The routine at #069F7h recalculates this value using the addresses stored in #70579h and #70574h (see the earlier descriptions of these two addresses for more information).

#70673h	Next error to display		5 nibbles
#70678h			1 nibble
#70679h	ATTN flag		5 nibbles
#7067Eh			33 nibbles
#7069Fh	Stack size		5 nibbles
#706A4h	Random number seed		16 nibbles
#706B4h			15 nibbles
#706C3h	Annunciators		2 nibbles
#706C5h	System	*Flags*	16 nibbles
#706D5h	User		16 nibbles
#706E5h			26 nibbles
#706FFh	Error number		5 nibbles
#70704h			15 nibbles
#70713h	Prolog	*GROB of character under the cursor*	5 nibbles
#70718h	Length		5 nibbles
#7071Dh	Height (6)		5 nibbles
#70722h	Width (10)		5 nibbles
#70727h	Pixels		20 nibbles
#7073Bh			142 nibbles
#707C9h	Current menu offset		5 nibbles
#707CEh	Last menu offset		5 nibbles
#707D3h			6 nibbles
#707D9h	Number of attached libraries		3 nibbles
#707DCh	Number	*First library info.*	3 nibbles
#707DFh	@ of info.		5 nibbles
	⋮		
	Number	*Last library info.*	3 nibbles
	@ of info.		5 nibbles

Next Error to Display

#70673h is used to store the number of the next error message to be displayed. When the calculator returns to interactive mode, this address is checked to see if a message is waiting. If so, then the error displayed.

Attn Flag

The five nibbles at #70679h are set to 0 if the [ON] key has not been pressed. Otherwise, they contain the number of times that the key was pressed. These five nibbles are used by machine language programs (such as BEEP) to know if they must stop execution.

Stack Size

At #7069Fh is the stack size, measured in nibbles. The stack always contains at least 5 zero nibbles, so the stack size is equal to 5*(DEPTH+1).

Random Number Seed

At #706A4h is a random number seed used by the RAND function. This seed is a "real" object minus the prolog. RDZ is a function that can change the value of the seed.

Annunciators

The two nibbles at #706C3h contain the current state of the HP 48's annunciators. If a bit is set, then the corresponding annunciator is showing:

Flags

These flags are stored in #706C5h and #706E4h, as shown opposite.

System Flags (-1 to -64):

	Bit 3	Bit 2	Bit 1	Bit 0
#706C5h	-4	-3	-2	-1
#706C6h	-8	-7	-6	-5
#706C7h	-12	-11	-10	-9
#706C8h	-16	-15	-14	-13
#706C9h	-20	-19	-18	-17
#706CAh	-24	-23	-22	-21
#706CBh	-28	-27	-26	-25
#706CCh	-32	-31	-30	-29
#706CDh	-36	-35	-34	-33
#706CEh	-40	-39	-38	-37
#706CFh	-44	-43	-42	-41
#706D0h	-48	-47	-46	-45
#706D1h	-52	-51	-50	-49
#706D2h	-56	-55	-54	-53
#706D3h	-60	-59	-58	-57
#706D4h	-64	-63	-62	-61

User Flags (1 to 64):

	Bit 3	Bit 2	Bit 1	Bit 0
#706D5h	4	3	2	1
#706D6h	8	7	6	5
#706D7h	12	11	10	9
#706D8h	16	15	14	13
#706D9h	20	19	18	17
#706DAh	24	23	22	21
#706DBh	28	27	26	25
#706DCh	32	31	30	29
#706DDh	36	35	34	33
#706DEh	40	39	38	37
#706DFh	44	43	42	41
#706E0h	48	47	46	45
#706E1h	52	51	50	49
#706E2h	56	55	54	53
#706E3h	60	59	58	57
#706E4h	64	63	62	61

Error Number

#706FFh stores the number of the last error that occurred. This number is set to #00000h if no error is saved; it is set to #70000h if the error message was one defined by the user. A list of all error messages and their numbers is given in the appendix.

GROB of the Character Under the Cursor

Starting at #70713 is a graphics object that is used to remember the character underneath the cursor during edit mode.

Menu Offsets

These two sets of 5 nibbles each at #707C9h and #707CEh contain the offsets for the menu display (that is, the number of the first menu label to display). For more information, see the explanation of the addresses #7061Eh and #70623 on page 198.

Number of Attached Libraries

The 3 nibbles at #707D9h contain the number of attached libraries. Each of these libraries is described by its number, followed by the address where the library information is stored.

If the information is found in hidden ROM, then the address points to a system binary (located in accessible memory) that contains the address in hidden ROM. In every case, the address that points to the library's declaration is found immediately after the name, at @+n_c*2+Eh (using the same notation as that in **Chapter 11**, page 143).

This library beginning contains all the necessary information for retrieving the contents of the library (messages, commands, etc.). In particular , it makes it easy to find the error messages, knowing that the number of such a message has two parts: the library number in which it is stored (3 nibbles), and its order number in the message table (2 nibbles — a library can therefore have a maximum of 256 messages). The message number is

*Library number**256+*order number*

Using only an error number, we can easily determine the corresponding library number. The list of attached libraries can then be used to find the message table starting address which contains the error text.

It is possible to modify this information table, and then completely rewrite the HP 48's error messages. This could be very useful for translating all the error messages to another language, for example.

Conclusion

The reserved memory area normally ends at #70844h, but it can be extended, if necessary. For example, some ROM cards, like the HP solver card, reserve some extra memory (for new libraries, among other things).

This description of RAM is not complete, but it contains the majority of useful items necessary for the machine language programmer who wishes to create programs that need access to the HP 48's resources.

15. Programming in Machine Language

In the preceding chapters, we have studied the internal functionality of the HP 48. We will now use this knowledge to access all the machine's resources, particularly for programming in machine language. The HP 48 can handle only objects, so we will use the Code object (see **Chapter 11**) to contain a machine language program.

The problem is in creating this object. Using a more general approach, we will see how to create any type of object. We have seen that any object can be represented by a series of hexadecimal digits. We will write a function to transform a sequence of hexadecimal digits into the corresponding object. The user will simply enter a string of characters containing the digits to be transformed into a corresponding series of nibbles.

In a string, characters are stored using their ASCII code. For example, the hexadecimal digit A is 10 in decimal, and is stored as #41h in ASCII. There is a simple object that consists of hexadecimal digits when edited but is stored as nibbles in memory. This object is the GROB, or graphics object. The transformation from hex digits to nibbles will be done using this object.

The GROB has the following structure:

@	Prolog (02B1E)		5 nibbles
@+5h	Total length excluding prolog l_t		5 nibbles
@+Ah	Number n_l of lines (in pixels)		5 nibbles
@+Fh	Number n_c of columns (in pixels)		5 nibbles
@+14h	Columns 1 to 8	*Pixels in line 1*	1+1 nibbles
	⋮		
	Last pixels		1+1 nibbles
	⋮	⋮	
	Columns 1 to 8	*Pixels in line n_l*	1+1 nibbles
	⋮		
	Last pixels		1+1 nibbles
@+l_t+5h			

We can see that the HP 48 uses blocks of 8 columns. We will therefore create a graphics object with 8 columns and the number of lines will be equal to the number of hexadecimal digits (of our code) divided by 2 (8 pixels take up 2 nibbles, therefore 2 hexadecimal digits). If the number of hexadecimal digits is odd, we will round it up after the division. In this manner, the memory occupied by the GROB (excluding the prolog, length, and size information) will be, at the most, the number of hexadecimal digits, plus one (in nibbles). This coding can be done with this sequence:

```
"GROB 8 " OVER SIZE 2 / CEIL + " " + SWAP + OBJ→
```

This prepares the graphics object in a string in the following manner:

- The beginning of the GROB is placed in a string ("GROB 8 ");
- We calculate the number of lines in the GROB with OVER SIZE 2 / CEIL and we add it to the first part of the GROB;
- Next, we add the list of hexadecimal digits (separating it from the rest with the addition of " ") by " " + SWAP +;
- And, finally, we transform the string of characters into a graphics object by the command OBJ→.

We can simplify this program slightly by removing the CEIL command (which is done automatically when the string is transformed via `OBJ→`). We now have `"GROB 8 " OVER SIZE 2 / + " " + SWAP + OBJ→` This places a graphics object on the stack for the object that we want to create. Now, in memory is the following structure:

@	Prolog (02B1E)	5 nibbles
@+5h	Total length excluding prolog l_t	5 nibbles
@+Ah	Number n_l of lines (in pixels)	5 nibbles
@+Fh	Number n_c of columns (in pixels)	5 nibbles
@+14h	Object to be created	l_t-15 nibbles
@+l_t+5h		

We know that only addresses are stored on the stack. To access the object we want to create, we need only take the address, @, of the GROB on the stack and replace it with @+14h. This removes the prolog, length, number of columns, and number of lines. There is a SYSEVAL call that will perform this function. The call to #056B6h takes a system binary as an argument which contains the number of 5 nibble blocks to remove and returns the new object as well as an "external" which is not useful here. W e need to remove 4 blocks of 5 nibbles, so we need a system binary equal to 4. Such an object is stored at #04017h. Therefore, the transformation from GROB to object can be done by: `#4017h SYSEVAL #56B6h SYSEVAL DROP` The first SYSEVAL recalls the system binary to the stack, and the second SYSEVAL performs the transformation. The last thing to do is to recreate the object in such a way that the pointer to it (on the stack) is really pointing to the object itself, and not its contents. This is done easily with the NEWOB function which recreates the object in level 1 of the stack, and modifies all necessary pointers.

We now have the final version of the program GASS (Graphic ASSembler):

```
GASS (# 1DB3h)
  « "GROB 8 " OVER SIZE 2 / + " " + SWAP + OBJ→
    #4017h SYSEVAL #56B6h SYSEVAL DROP NEWOB
  »
```

This program is quite fast; the transformation from hexadecimal digits to nibbles is done by machine language routines found in ROM. However , those routines also perform verifications and calculations that slow down the process a little. A faster version of GASS, written entirely in machine language, is given in the **Library of Programs** (called RASS).

Let's try this program to create a small object. *(Note: To make this code more readable, it is presented in blocks of 5 digits, but these spaces are not part of the code. You must enter this code in a contiguous manner — no spaces, no new lines).* Here is the code listing for a small object:

```
C2A20 B1000 7556C 6C602 46F6E 65602 12
```

To code this object, just enter the code as a character string (with no spaces, no new lines): `"C2A20B10007556C6C60246F6E6560212"` Then execute GASS. A couple of seconds later, the object is on the stack.

Now that you know how to create any object, you can see how to create machine language programs. In writing such programs, you should always remember these important points:

- *The contents of certain registers*:
  - **D0** is the pointer to the next object to be executed (after the machine language program). To continue to the next object after the machine language program has finished, do this: `A=DAT0 A,D0=D0+5,PC=(A)` (coded as `142164808C`).
  - **D1** is the stack pointer. If we execute `A=DAT1 A`, field **A** of register **A** contains the address of the object in level 1. If we increment **D1** by 5 (`D1=D1+5`) then we move to level 2 (at this point, the instruction `A=DAT1 A` will place the address of the object in level 2 into **A** field **A**).
  - **B** contains the address of the return stack end—not too useful.
  - **D** contains the amount of free memory in number of 5 nibble blocks (the same size as the stack levels).

    Unless you intend to change them, these 4 registers must be restored to their original values before ending the program via `142164808C`. To restore them, here are 2 useful routines:
  - SAVE_REG, at address #0679Bh (called with a GOSBVL #0679B) saves these registers in the reserved RAM.
  - LOAD_REG, at address #067D2h (called with a GOSBVL #067D2) restores the register values previously saved.
- *The structures of the objects*: To take an object from the stack, you must know its internal structure to handle it properly. Also, including HP 48 objects in your program lets you profit from the RPL functions.
- *The RAM structure*: This is a must if you ever need to access RAM.

You can also call routines found in ROM (e.g. `SAVE_REG` and `LOAD_REG`). One of the best exercises in applying **Part Two** is to analyze the machine lan-guage programs in the **Library of Programs**, or to disassemble certain routines in ROM.

The next step is to write your own machine language programs. Start with simple ideas. For example, to test the speed of machine language programs, you might compare the execution speeds of two programs, one in machine language, one in RPL. This test could be two programs that simply count to 1000 (`1 1000 START NEXT`).

Part Three:

Library of Programs

Notice

This **Library of Programs** contains numerous utilities written in machine language. In most cases they can be used without any specific knowledge, except for the method used to enter them.

To make the code more readable, the machine language programs (which consist of hexadecimal digits 0...9, A...F) are presented in groups of 5 digits separated by spaces. For example, the program NOTHING (which does nothing) would be presented in the form:

```
NOTHING(# B6F7h)
  CCD20 F0000 14216 4808C
```

To type in this program you would do the following:

- Enter the code as a character string *with no spaces and no new lines* (in this example, it would be "CCD20F0000142164808C").
- After verifying that the checksum given in parenthesis is correct, (this step is optional, but strongly recommended), execute the program GASS (or RASS once you have entered it) on the string. GASS (or RASS) returns the desired object to the stack. In the case of a machine language program, this is a "code" object, or a list of instructions that the machine can understand. Note:
  - To calculate the checksum, place the object on the stack and execute BYTES. This returns the object's checksum and size.
  - Use hexadecimal mode (execute HEX) to make the checksum comparisons; all checksums are given in hexadecimal.
  - The checksum for a machine language program is given for the character string before executing GASS (or RASS).
  - The program ALLBYTES will rapidly calculate all the checksums for a directory.
  - The presence of libraries containing commands with the same name as the programs used (or a similar name) may result in a checksum that is incorrect, even if the program is correct.
- The stack may now contain an unfamiliar object (shown by the word Code). *This object must never be edited*—doing so may destroy it. Just store it into a variable name (in this example: 'NOTHING' STO).

To assist you in checking for errors, we have included two programs:

- BY5 alters the character string to look like the form presented in this book (groups of 5 digits, 8 groups per line).
- CLEAN cleans a character string by removing all characters other than hexadecimal digits. CLEAN is written partially in machine language for speed.

One other note: Some programs contain the character "■". This symbol represents a carriage return, obtained by pressing the keys [→][·].

To summarize: Before typing any machine language programs, you will need to enter the two RPL programs GASS and BY5. You should practice entering an assembly program by entering NOTHING (which is quite short, and thus less likely that you will make a mistake), then enter the program CLEAN.

At this point, you have the tools necessary to access all of your HP 48's resources that have been revealed in this book.

GASS

GASS is a program used to create objects. It can create any object from a listing of hexadecimal codes. GASS is explained in detail in **Chapter 15**. It takes a character string containing a series of hexadecimal codes from the stack, and returns the corresponding object.

```
GASS (# 1DB3h)
  «
    "GROB 8 " OVER SIZE 2 / + " " + SWAP + OBJ→
    #4017h SYSEVAL #56B6h SYSEVAL DROP NEWOB
  »
```

Note: Creating objects is an operation that you must perform with caution. You must not transform just any list of codes, only lists which contain valid objects. Therefore, you should carefully verify the character strings before executing GASS.

ALLBYTES

The program ALLBYTES calculates the checksum for all objects contained in the current directory. It returns a character string which contains the names of each object followed by its checksum (in hexadecimal).

```
ALLBYTES(# 52FFh)
  «
    VARS
    → V
    «
      HEX "■" 1 V SIZE
      FOR X
        V X GET SWAP OVER →STR 2 OVER SIZE 1 -
        SUB ":" + "             " OVER SIZE
        15 SUB + + SWAP BYTES DROP "■" + +
      NEXT
    »
  »
```

(There are 13 spaces in the text string in the eighth line of the above program.)

BY5

BY5 is a small utility to change character strings into a more readable form. This form is identical to that used in this book (groups of 5 digits, 8 groups per line).

BY5 is very useful as you look through your code for errors detected by the checksum. For example,

```
"CCD20F0000142164808C" BY5
```

returns `"CCD20 F0000 14216 4808C "`

BY5 (# 74BAh)

```
«
   → S
   «
     "■" 0 S SIZE 1 -
     FOR X
       1 40
       FOR Y
         S X Y + DUP 4 + SUB + " " + 5
       STEP
       "■" + 40
     STEP
   »
»
```

CLEAN

CLEAN is the inverse function of BY5: It removes all characters from a string that are not hexadecimal digits (0...9, A...F). It prepares a string for the program GASS, after using BY5 to check for errors.

This program is written partially in machine language, so it must be entered according to the specifications given on pages 213-214.

Here is the commented assembly source listing for CLEAN:

```
      D9D20    CON(5)   PROL_PRGM        Program object
      B4E02    CON(5)   STRING_SPC       " "
      76BA1    CON(5)   ADD              +
      CCD20    CON(5)   PROL_CODE        Code object
start 08000    CON(5)   (end)-(start)    Code length
      8FB9760  GOSBVL   SAVE_REG         Bckup regs.
      143      A=DAT1   A                A=size
      130      D0=A                      D0=D1=address
      131      D1=A                        object in level 1
      169      D0=D0+   10               D0=Backup Regs.
      174      D1=D1+   5                D1=contents addr.
      143      A=DAT1   A
      174      D1=D1+   5
      818F84   A=A-5    A
      819F0    ASRB     A
      D8       B=A      A                B=# of characters
                                           in the string
l1    8A9      ?B=0     A                Done?
      83       GOYES    l4               Yes --> end!
      14B      A=DAT1   B
      3103     LCHEX    30               ASCII code for 0
      9E2      ?A<C     B
      32       GOYES    l3               Bad character
      3193     LCHEX    39               ASCII code for 9
      9EA      ?A<=C    B
      41       GOYES    l2               Good character
      3114     LCHEX    41               ASCII code for A
      9E2      ?A<C     B
      11       GOYES    l3               Bad character
      3164     LCHEX    46               ASCII code for F
```

```
        9E6      ?A>C     B
        80       GOYES    13             Bad character
l2      148      DAT0=A   B              Good char --> rewrite
        161      D0=D0+   2              Next
l3      171      D1=D1+   2
        CD       B=B-1    A              One less
        68CF     GOTO     l1             Loop again
l4      AE0      A=0      B              Mark the end
        148      DAT0=A   B                with char 00
        8F2D760  GOSBVL   LOAD_REG       Restore regs.
        142      A=DAT0   A              Return to RPL
        164      D0=D0+   5
        808C     PC=(A)
end
        9C2A2    CON(5)   REAL_1
        92CF1    CON(5)   OVER
        C2A20    CON(5)   PROL_STRING    ;
        70000    CON(5)   #00007         ;CHR 00
        00       CON(2)   #00            ;
        4BAC1    CON(5)   POS
        9C2A2    CON(5)   REAL_1
        90DA1    CON(5)   MINUS
        C58C1    CON(5)   SUB
        B2130    CON(5)   EPILOG
```

CLEAN (# CD56h)

```
D9D20 B4E02 76BA1 CCD20 08000 8FB97 60143 13013
11691 74143 17481 8F848 19F0D 88A98 314B3 1039E
23231 939EA 41311 49E21 13164 9E680 14816 1171C
D68CF AE014 88F2D 76014 21648 08C9C 2A292 CF1C2
A2070 00000 4BAC1 9C2A2 90DA1 C58C1 B2130
```

PEEK

PEEK allows you to look at the memory contents at a specific address. Simply give it an address and the number of bytes to read, and it will return a character string with the hexadecimal code that was read. For example, #0 #5 PEEK returns the first 5 nibbles of the HP 48 ROM: "2369B".

PEEK does not offer access to the hidden ROM (ROM area at #70000h). To access that area, use the program HRPEEK (Hidden ROM PEEK).

Here is the commented assembly source listing for PEEK:

```
      D9D20    CON(5)  PROL_PRGM        Program object
      2ABF1    CON(5)  DUP2             Verify the number
      3FBF1    CON(5)  DROP2              of arguments
      CCD20    CON(5)  PROL_CODE        Code object
start 3A000    CON(5)  (end)-(start)    Code length
      8FB9760  GOSBVL  SAVE_REG         Backup regs.
      147      C=DAT1  A
      134      D0=C
      169      D0=D0+  10               D0=address of contents
                                          of object in stack level
                                          1 (the PEEK length)
      142      A=DAT0  A                Read # of nibbles to read
      340FFF7  LCHEX   #7FFF0           Maximum size
      8B6      ?C<A    A
      40       GOYES   l0               Size correct
      D6       C=A     A                Size too big—set to max.
l0    C6       C=C+C   A                Number of nibbles to
                                        reserve (2 per character)
      8FD7B50  GOSBVL  #05B7D           Reserve
      132      AD0ex
      147      C=DAT1  A
      134      D0=C                     D0=address of object in
                                          stack level 1
      169      D0=D0+  10
      146      C=DAT0  A                Read the contents (size
                                          to peek)
      D5       B=C     A
      174      D1=D1+  5
      147      C=DAT1  A
```

```
     134       D0=C                       D0=address of object in
                                             stack level 2
     169       D0=D0+    10
     146       C=DAT0    A                Read the contents
     135       D1=C
     130       D0=A
l1   8A9       ?B=0      A                Done?
     F2        GOYES     l3               Yes --> end
     AE0       A=0       B
     15B0      A=DAT1    1                Read one nibble
     3103      LCHEX     #30              ;
     A6A       A=A+C     B                ; Transform
     3193      LCHEX     39               ; to ASCII code
     9EA       ?C>=A     B                ; (0->'1'=48...
     90        GOYES     l2               ; 15->'F'=70)
     3170      LCHEX     07               ;
     A6A       A=A+C     B                ;
l2   148       DAT0=A    B                Write into the string
     161       D0=D0+    2                Next character
     170       D1=D1+    1                Next nibble
     CD        B=B-1     A                One less
     61DF      GOTO      l1               Loop
l3   8F2D760   GOSBVL    LOAD_REG         Restore regs.
     174       D1=D1+    5                DROP
     E7        D=D+1     A
     118       C=R0
     145       DAT1=C    A                Result -> stack
     142       A=DAT0    A                Return to RPL
     164       D0=D0+    5
     808C      PC=(A)
end  B2130     CON(5)    EPILOG           Program end
```

PEEK (# ED02h)

```
D9D20 2ABF1 3FBF1 CCD20 3A000 8FB97 60147 13416
91423 40FFF 78B64 0D6C6 8FD7B 50132 14713 41691
46D51 74147 13416 91461 35130 8A9F2 AE015 B0310
3A6A3 1939E A9031 70A6A 14816 1170C D61DF 8F2D7
60174 E7118 14514 21648 08CB2 130
```

POKE

POKE is the inverse of PEEK. It will write data to a specific address. As arguments, it takes a binary integer (the address), in level 2, and a series of hexadecimal digits (the data), in level 1.

CAUTION: Use this program carefully! You can corrupt memory and disturb the normal functionality of the HP 48 with this program. However, the programs in this book that use POKE can be used with no danger.

Here is the commented assembly source listing for POKE:

```
      D9D20    CON(5)   PROL_PRGM       Program Object
      2ABF1    CON(5)   DUP2            Verify the number
      3FBF1    CON(5)   DROP2             of arguments
      CCD20    CON(5)   PROL_CODE       Code object
start 48000    CON(5)   (end)-(start)   Code length
      8FB9760  GOSBVL   SAVE_REG        Backup regs.
      143      A=DAT1   A
      132      AD0ex                    D0=address of object in
                                          stack level 1
      164      D0=D0+   5
      146      C=DAT0   A               C=length (5+2*number
                                          of characters in string)
      164      D0=D0+   5
      D5       B=C      A
      174      D1=D1+   5
      143      A=DAT1   A
      131      D1=A                     D1=address of object 2
                                          (poke address)
      179      D1=D1+   10
      143      A=DAT1   A
      131      D1=A                     D1=address of where to
                                          poke
      3450000  LCHEX    #00005
l1    E1       B=B-C    A
      8A9      ?B=0     A               Done?
      13       GOYES    13              Yes -> end
      14A      A=DAT0   B               Read a char
```

```
     3103     LCHEX   30          ;
     B6A      A=A-C   B           ;Convert ASCII
     3190     LCHEX   09          ;to Hexadecimal
     9EA      ?C>=A   B           ;(48='0' -> 0
     90       GOYES   l2          ;70='F' -> 15)
     3170     LCHEX   07          ;
     B6A      A=A-C   B           ;
l2   1590     DAT1=A  1           Write to memory
     161      D0=D0+  2           Next char
     170      D1=D1+  1           Next nibble
     3420000  LCHEX   #00002
     6DCF     GOTO    l1          Loop...
l3   8F2D760  GOSBVL  LOAD_REG    Restore regs.
     179      D1=D1+  10          ;
     E7       D=D+1   A           ;DROP2
     E7       D=D+1   A           ;
     142      A=DAT0  A           Return to RPL
     164      D0=D0+  5
     808C     PC=(A)
end  B2130    CON(5)  EPILOG      Program end
```

POKE (# 14A5h)

```
D9D20 2ABF1 3FBF1 CCD20 48000 8FB97 60143 13216
41461 64D51 74143 13117 91431 31345 0000E 18A91
314A3 103B6 A3190 9EA90 3170B 6A159 01611 70342
00006 DCF8F 2D760 179E7 E7142 16480 8CB21 30
```

HRPEEK

HRPEEK allows you to read the contents of the hidden ROM, which is normally not accessible. In order to do this, HRPEEK must calculate its own address (either in built-in RAM, or in a plug-in card), and then displace the built-in RAM at #70000h to allow access to the hidden ROM (#70000h to #7FFFFh). By calculating its own address, HRPEEK will be able to tell whether or not it is affected by this memory displacement.

HRPEEK is generally the same as PEEK, and the argument syntax is the same. For example, the command #70000h #10h HRPEEK (peek at 16 nibbles starting at #70000h in the hidden ROM) will return the character string "D21098FFFB108E78".

CAUTION: You should not use HRPEEK to peek at any memory location except (#70000h - #7FFFFh) or you may get data that is invalid. This is because of the built-in memory displacement that must take place.

One other note: As HRPEEK displaces the built-in RAM, the screen will show a little "static" during the execution of the program. This is normal and you need not worry about it.

Here is the commented assembly source for HRPEEK:

```
      D9D20    CON(5)   PROL_PRGM        Program object
      2ABF1    CON(5)   DUP2             Verify the number
      3FBF1    CON(5)   DROP2               of arguments
      CCD20    CON(5)   PROL_CODE        Code object
start D4100    CON(5)   (end)-(start)    Code length
      8FB9760  GOSBVL   SAVE_REG         Backup regs.
      147      C=DAT1   A
      134      D0=C                      D0=address of object in
                                            stack level 1
      169      D0=D0+   10               D0=address of object
                                            contents in stack level
                                            1 (PEEK length)
      142      A=DAT0   A                Read number of nibbles
                                            to be read
      340FFF7  LCHEX    #7FFF0           Maximum size
```

```
      8B6       ?C<A     A
      40        GOYES    l1           Size correct
      D6        C=A      A            Size is too big—change
                                        to maximum.
l1    C6        C=C+C    A            No. of nibbles to reserve
                                        (2 per character)
      8FD7B50   GOSBVL #05B7D         Reserve
      132       AD0ex
      147       C=DAT1   A
      134       D0=C                  D0=address of object in
                                        stack level 1
      169       D0=D0+   10
      146       C=DAT0   A            Read the contents (size
                                        of peek)
      10C       R4=C
      174       D1=D1+   5
      147       C=DAT1   A
      134       D0=C                  D0=address of object in
                                        stack level 2
      169       D0=D0+   10
      146       C=DAT0   A            Read the contents
      10A       R2=C
      103       R3=A
      84F       ST=0     15           No keyb. int.
l2    11C       C=R4
      8AE       ?C#0     A            Done?
      60        GOYES    l3
      62D0      GOTO     l8           Yes --> end
l3    CE        C=C-1    A            One less
      10C       R4=C
      808F      INTOFF
      81B4      A=PC     A            A=mem. address of 'here'
here  346CFF7   LC(5)    #80000-(l5)+(here)
      8BE       ?C<=A    A            where is HRPEEK ?
      03        GOYES    l5           In a plug-in card
      3482000   LC(5)    (l4)-(here)
      C2        C=C+A    A            C=memory address of 'l4'
      2F        P=       15
      80C4      C=P      4            C=address of 'l4' after dis-
                                        placement of built-in
                                        RAM to #F0000h
      20        P=       0
      8FFB620   GOSBVL   #026BF       Displace built-in RAM and
                                        call routine found at
                                        address in field A of C
```

```
        6160     GOTO   16
14      112      A=R2                 ;
        131      D1=A                 ;Read one nibble
        AE0      A=0        B         ;from R2 and
        15B0     A=DAT1     1         ;save it in
        101      R1=A                 ;register R1
        01       RTN                  ;
15      3400007  LCHEX      #70000    ;
        804      UNCNFG               ;
        340000F  LCHEX      #F0000    ;Displace the RAM
        805      CONFIG               ;to #60000h
        3400006  LCHEX      #60000    ;
        805      CONFIG               ;
        112      A=R2
        131      D1=A
        AE0      A=0        B
        15B0     A=DAT1     1         Read one nibble
        101      R1=A
        3400006  LCHEX      #60000    ;
        804      UNCNFG               ;
        340000F  LCHEX      #F0000    ;Return RAM
        805      CONFIG               ;to #70000h
        3400007  LCHEX      #70000    ;
        805      CONFIG               ;
16      8080     INTON                Interrupts OK
        111      A=R1
        3103     LCHEX      30        ;
        A6A      A=A+C      B         ;
        3193     LCHEX      39        ;Convert the
        9EA      ?C>=A      B         ;nibble read to
        90       GOYES      17        ;ASCII
        3170     LCHEX      07        ;
        A6A      A=A+C      B         ;
17      11B      C=R3
        134      D0=C
        148      DAT0=A     B         Write
        161      D0=D0+     2
        136      CD0ex
        10B      R3=C
        11A      C=R2
        E6       C=C+1      A         Next!
        10A      R2=C
        682F     GOTO       12        Loop
18      85F      ST=1       15
        8F2D760  GOSBVL     LOAD_REG  Restore regs.
```

```
    174     D1=D1+   5          ;
    E7      D=D+1    A          ; DROP
    118     C=R0
    145     DAT1=C   A          Resulting string on stack
    142     A=DAT0   A          Return to RPL
    164     D0=D0+   5
    808C    PC=(A)
end B2130   CON(5)   EPILOG     Program end
```

HRPEEK(# 4305h)

```
D9D20 2ABF1 3FBF1 CCD20 35100 8FB97 60147 13416
91423 40FFF 78B64 0D6C6 8FD7B 50132 14713 41691
4610C 17414 71341 69146 10A10 384F1 1C8AE 6062D
0CE10 C808F 81B43 46CFF 78BE0 33482 000C2 2F80C
4208F FB620 61601 12131 AE015 B0101 01340 00078
04340 000F8 05340 00068 05112 131AE 015B0 10134
00006 80434 0000F 80534 00007 80580 80111 3103A
6A319 39EA9 03170 A6A11 B1341 48161 13610 B11AE
610A6 82F85 F8F2D 76017 4E711 81451 42164 808CB
2130
```

?ADR

This program finds the address of the object in level 1 of the stack. Here is the commented assembly source listing of ?ADR:

```
      D9D20   CON(5)  PROL_PRGM       Program object
      E4A20   CON(5)  PROL_INT        Null binary integer where
      A0000   CON(5)  #0000A             the address will be
      00000   CON(5)  #00000
      CB2A1   CON(5)  NEWOB           Recreate binary integer
      DBBF1   CON(5)  SWAP
      CCD20   CON(5)  PROL_CODE       Code object
start 62000   CON(5)  (end)-(start)   Code length
      147     C=DAT1  A               C=@ of object
      174     D1=D1+  5               Remove object from
      E7      D=D+1   A                  stack
      143     A=DAT1  A
      133     AD1ex
      179     D1=D1+  10
      145     DAT1=C  A               Write @
      131     D1=A
      142     A=DAT0  A               Return to RPL
      164     D0=D0+  5
      808C    PC=(A)
end   B2130   CON(5)  EPILOG          Program end
```

```
?ADR (# 26A0h)
  D9D20 E4A20 A0000 00000 CB2A1 DBBF1 CCD20 62000
  14717 4E714 31331 79145 13114 21648 08CB2 130
```

SSAG

This program returns the hexadecimal codes of the object in level 1 of the stack. It performs the inverse of GASS (thus, the name SSAG). SSAG uses the programs PEEK and ?ADR.

To determine the size of the object, SSAG uses the SYSEVAL call #1A1FC which is the same function as BYTES, except it works with any object given as an argument. When BYTES is executed with a local name as an argument, for example, it returns the checksum and length of the contents of this name. The object on the stack is first stored in a global variable called 'OBJ.TMP' in order to assign it a fixed address.

Example: "123" SSAG would return "C2A20B0000132333" which is the code for a string object containing 3 characters: "1", "2", and "3" (ASCII codes #31h, #32h and #33h).

SSAG was written by Dominique Moisescu.

```
SSAG (# B7AFh)
  «
    'OBJ.TMP' STO 'OBJ.TMP' RCL DUP ?ADR SWAP
     # 1A1FCh SYSEVAL SWAP DROP 2 * R→B PEEK 'OBJ.TMP'
     PURGE
  »
```

RASS

RASS is the same as GASS, only it is written completely in machine language. Here is the commented assembly source listing for RASS:

```
      D9D20    CON(5)   PROL_PRGM       Program object
      78BF1    CON(5)   DUP             ;Verify there is
      8DBF1    CON(5)   DROP            ;at least one
                                        ;argument on stack
      CCD20    CON(5)   PROL_CODE       Code object
      BA000    CON(5)   (end)-(start)   Code length
start 8FB9760  GOSBVL   SAVE_REG        Backup regs.
      147      C=DAT1   A
      137      CD1ex                    D1=string address
      109      R1=C
      174      D1=D1+   5
      143      A=DAT1   A               A=string length
      3450000  LCHEX    #00005
      8A2      ?C=A     A               Empty string?
      57       GOYES    15              Yes --> end
      EA       A=A-C    A
      81C      ASRB                     Number of codes
      103      R3=A
      174      D1=D1+   5
      137      CD1ex
      10A      R2=C
      D6       C=A      A
      84A      ST=0     10
l1    8F8DA60  GOSBVL   #06AD8          Reserve memory
      501      GONC     12              Ok!
      8FD3361  GOSBVL   #1633D          Garbage collector
      11B      C=R3
      6BEF     GOTO     11
l2    119      C=R1
      135      D1=C
      132      AD0ex
      141      DAT1=A   A               Object reserved on
                                          the stack
      130      D0=A
      113      A=R3
      D8       B=A      A
      CD       B=B-1    A
      11A      C=R2
      135      D1=C
```

```
l3   14B      A=DAT1   B          read one code
     3103     LCHEX    #30        ;
     B6A      A=A-C    B          ;
     3190     LCHEX    #09        ;ASCII Code
     9EA      ?C>=A    B          ;-> hexadecimal
     90       GOYES    l4         ;
     3170     LCHEX    #07        ;
     B6A      A=A-C    B          ;
l4   1580     DAT0=A   1          Write
     160      D0=D0+   1
     171      D1=D1+   2
     CD       B=B-1    A          One less
     59D      GONC     l3         Continue if necessary
l5   8F2D760  GOSBVL   LOAD_REG   Restore regs.
     142      A=DAT0   A          Return to RPL
     164      D0=D0+   5
     808C     PC=(A)
end  B2130    CON(5)   EPILOG     Program end
```

```
RASS (# B5D3h)
  D9D20 78BF1 8DBF1 CCD20 BA000 8FB97 60147 13710
  91741 43345 00008 A257E A81C1 03174 13710 AD684
  A8F8D A6050 18FD3 36111 B6BEF 11913 51321 41130
  113D8 CD11A 13514 B3103 B6A31 909EA 90317 0B6A1
  58016 0171C D59D8 F2D76 01421 64808 CB213 0
```

CHK

This program checks the number of objects on the stack, and their type. It is not interesting by itself, but it is extremely useful for a programmer who needs to check that the correct arguments were passed to his program.

CHK takes two binary integers from the stack. The first argument (stack level 2) is the number of arguments—from 0 (meaning no arguments) to 8. The other argument is the type description. Each type is represented by a two digit hexadecimal number, as shown in the table below. If the arguments passed to CHK are bad (i.e. number of arguments larger than 8, or an invalid type), you'll get an error: Too Few Arguments or Bad Argument Value. If the arguments are valid, nothing will happen; the arguments will disappear. Examples: To verify that the stack contains...

- a character string and another object of any type: #2 #0900h CHK
- two binary integers: #2h #0A0Ah CHK
- eight objects of any type: #8h #0h CHK
- a global name and two real numbers: #3h #1A0202h CHK

Prolog	Object Type	Code
	Any Object	00
02911	System Binary	01
02933	Real Number	02
02955	Long Real	03
02977	Complex Number	04
0299D	Long Complex	05
029BF	Character	06
029E8	Array	07
02A0A	Linked Array	08
02A2C	String	09
02A4E	Binary Integer	0A
02A74	List	0B
02A96	Directory	0C
02AB8	Algebraic Object	0D
02ADA	Unit Object	0E
02AFC	Tagged Object	0F
02B1E	Graphic Object	10

Prolog	Object Type	Code
02B40	Library	11
02B62	Backup Object	12
02B88	Library Data	13
02BAA	Reserved 1	14
02BCC	Reserved 2	15
02BEE	Reserved 3	16
02C10	Reserved 4	17
02D9D	Program	18
02DCC	Code	19
02E48	Global Name	1A
02E6D	Local Name	1B
02E92	XLIB Name	1C

Here is the commented assembly source listing for CHK:

```
      CCD20    CON(5)   PROL_CODE     Code object
start 99100    CON(5)   (end)-(start) Code length
      8FB9760  GOSBVL   SAVE_REG      Backup regs.
      AF0      A=0      W             ;First verify:
      808202   LAHEX    #2            ;the arguments for
      AF2      C=0      W             ;CHK: two
      33A0A0   LCHEX    #0A0A         ;binary integers
      7270     GOSUB    chk           ;
      8F2D760  GOSBVL   LOAD_REG      Restore regs.
      179      D1=D1+   10            ;DROP the two
      E7       D=D+1    A             ;binary integers
      E7       D=D+1    A
      8FB9760  GOSBVL   SAVE_REG      Backup regs.
      1C9      D1=D1-   10
      3480000  LCHEX    #00008        Maximum arguments
      D5       B=C      A
      147      C=DAT1   A
      134      D0=C
      169      D0=D0+   10
      1567     C=DAT0   W             C(W)=types
      174      D1=D1+   5
      143      A=DAT1   A
      130      D0=A
      169      D0=D0+   10
      1527     A=DAT0   W
      174      D1=D1+   5
      8B0      ?A>B     A             More than 8 args?
      71       GOYES    err1          Yes --> error
```

```
     7820     GOSUB    chk        Verify
     8F2D760  GOSBVL   LOAD_REG   Restore regs.
     142      A=DAT0   A          Return to RPL
     164      D0=D0+   5
     808C     PC=(A)
err1 3430200  LCHEX    #00203     Error: Bad Arg. Value
     DAerr    A=C      A
     8F2D760  GOSBVL   LOAD_REG   Restore regs.
     8D32050  GOVLNG   #05023     Error
chk  7190     GOSUB    chk2       Finds starting address of
                                     after prologue listing
     00000    CON(5)   #00000     Any object.
     11920    CON(5)   #02911     System binary
     33920    CON(5)   #02933     Real number
     55920    CON(5)   #02955     Long real
     77920    CON(5)   #02977     Complex number
     D9920    CON(5)   #0299D     Long complex
     FB920    CON(5)   #029BF     Character
     8E920    CON(5)   #029E8     Array
     A0A20    CON(5)   #02A0A     Linked array
     C2A20    CON(5)   #02A2C     String
     E4A20    CON(5)   #02A4E     Binary integer
     47A20    CON(5)   #02A74     List
     69A20    CON(5)   #02A96     Directory
     8BA20    CON(5)   #02AB8     Algebraic object
     ADA20    CON(5)   #02ADA     Unit object
     CFA20    CON(5)   #02AFC     Tagged object
     E1B20    CON(5)   #02B1E     Graphics object
     04B20    CON(5)   #02B40     Library
     26B20    CON(5)   #02B62     Backup object
     88B20    CON(5)   #02B88     Library data
     AAB20    CON(5)   #02BAA     Reserved 1
     CCB20    CON(5)   #02BCC     Reserved 2
     EEB20    CON(5)   #02BEE     Reserved 3
     01C20    CON(5)   #02C10     Reserved 4
     D9D20    CON(5)   #02D9D     Program
     CCD20    CON(5)   #02DCC     Code
     84E20    CON(5)   #02E48     Global name
     D6E20    CON(5)   #02E6D     Local name
     29E20    CON(5)   #02E92     XLIB name
chk2 D8       B=A      A          Object number.
     AF7      D=C      W          Types
l1   07       C=RSTK              C=starting address of list
     8A9      ?B=0     A          Done?
     00       RTNYES              Yes
```

```
        134      D0=C
        06       RSTK=C                Backup
        31D1     LCHEX     #1D
        9E7      ?D<C      B           Type OK?
        60       GOYES     l2
        693F     GOTO      err1        No --> error
l2      96B      ?D=0      B           ;
        C0       GOYES     l3          ; Get a
        A6F      D=D-1     B           ; prolog
        164      D0=D0+    5           ;
        64FF     GOTO      l2          ;
l3      147      C=DAT1    A
        8AE      ?C#0      A
        D0       GOYES     l4
        3410200  LCHEX     #00201      End of stack --> error
        6E1F     GOTO      err            (Too Few Arguments)
l4      137      CD1EX
        143      A=DAT1    A           A=object prologue
        135      D1=C
        146      C=DAT0    A
        8AA      ?C=0      A           Any object?
        21       GOYES     l5          Yes --> OK
        8A2      ?A=C      A           Prologue OK?
        D0       GOYES     l5          Yes --> l5
        3420200  LCHEX     #00202      Obj. prologue≠required
        6DFE     GOTO      err         --> "Bad Argument Type"
l5      CD       B=B-1     A           One less
        BF7      DSR       W           Next type
        BF7      DSR       W
        174      D1=D1+    5           Next object
        689F     GOTO      l1          Loop
end
```

```
CHK (# FD7Ch)
  CCD20 99100 8FB97 60AF0 80820 2AF23 3A0A0 72708
  F2D76 0179E 7E78F B9760 1C934 80000 D5147 13416
  91567 17414 31301 69152 71748 B0717 8208F 2D760
  14216 4808C 34302 00DA8 F2D76 08D32 05071 90000
  00119 20339 20559 20779 20D99 20FB9 208E9 20A0A
  20C2A 20E4A 2047A 2069A 208BA 20ADA 20CFA 20E1B
  2004B 2026B 2088B 20AAB 20CCB 20EEB 2001C 20D9D
  20CCD 2084E 20D6E 2029E 20D8A F7078 A9001 34063
  1D19E 76069 3F96B C0A6F 16464 FF147 8AED0 34102
  006E1 F1371 43135 1468A A218A 2D034 20200 6DFEC
  DBF7B F7174 689F
```

REVERSE

The Saturn microprocessor writes all data to memory in reverse; you must reverse it to get the proper order. REVERSE reverses the characters in a string—which helpful for interpreting the data read by PEEK.

Example: "123" REVERSE returns "321".

Here is the commented assembly source listing for REVERSE:

```
      D9D20   CON(5)  PROL_PRGM       Program object
      FD550   CON(5)  #055DF          Empty string
      76BA1   CON(5)  add  +
      CCD20   CON(5)  PROL_CODE       Code object
start 86000   CON(5)  (end)-(start)   Code length
      8FB9760 GOSBVL  SAVE_REG        Backup regs.
      143     A=DAT1  A
      131     D1=A                    D1=string address
      174     D1=D1+  5
      137     CD1ex
      135     D1=C
      143     A=DAT1  A               A=string length
      C2      C=C+A   A
      134     D0=C
      174     D1=D1+  5               D1=address of first
                                         character
      181     D0=D0-  2               D0=address of last
                                         character
      818F84  A=A-5   A
      8A8     ?A=0    A               Empty string?
      52      GOYES   12              Yes --> end
l1    14B     A=DAT1  B               ;
      14E     C=DAT0  B               ;Switch two
      14D     DAT1=C  B               ;characters
      148     DAT0=A  B               ;
      171     D1=D1+  2
      181     D0=D0-  2
      133     AD1ex
      131     D1=A
      136     CD0ex
      134     D0=C
```

```
      8BA       ?C>=A    A           Again?
      FD        GOYES    l1
l2    8F2D760   GOSBVL   LOAD_REG    Restore regs.
      142       A=DAT0   A           Return to RPL
      164       D0=D0+   5
      808C      PC=(A)
end   B2130     CON(5)   EPILOG      Program end
```

REVERSE(# AA7Dh)

```
D9D20 FD550 76BA1 CCD20 86000 8FB97 60143 13117
41371 35143 C2134 17418 1818F 848A8 5214B 14E14
D1481 71181 13313 11361 348BA FD8F2 D7601 42164
808CB 2130
```

CRNAME

CRNAME is a program which can create any global name (including "strange" names that cannot be entered from the keyboard, or the names of existing functions). Here are two ideas for this program:

- Create variables under reserved names, which are then dif ficult to purge, visit, or change (giving them a certain security).
- Create variables with the same name as an HP 48 internal function in order to replace it. If the user types this name, then your program is executed rather than the internal function.

```
CRNAME (# 11E9h)
  «
    1 127 SUB 116 CHR 42 CHR + 128 CHR +
    228 CHR + 2 CHR + OVER SIZE CHR + SWAP
    + 43 CHR + 49 CHR + 0 CHR +
    # 4003h SYSEVAL # 56B6h SYSEVAL DROP NEWOB
    1 GET
  »
```

The principle of this program is the same as with GASS: a special object is created (here it's a string), which contains the desired object codes (the name in a list). Then certain information is stripped from the object to leave only the object contents.

We need to remove the prolog and the length of the string—2 blocks of 5 nibbles. The routine at #056B6h is used to take a system binary containing the number of 5 nibble blocks to be removed. This system binary exists in ROM (see the list of useful objects in ROM found in the appendix) at the address #04003h. It is recalled to the stack with #4003h SYSEVAL. After the NEWOB, a list containing the desired name is on the stack. The operation 1 GET removes the name from the list, and places it on the stack by itself.

CLVAR

The CLVAR instruction will purge all user variables in the current directory. This command can be executed with the press of three buttons ([→], [DEL], [ENTER]).

In the hands of an amateur, this can be very dangerous. It would, therefore, be wise to remove the access to this command. This can be done using the program CRNAME in the following manner:

- Enter any program. For example:

  `« "CLVAR Not Available!" DOERR »`

- Then type: `"CLVAR" CRNAME STO`

It is best to store this false CLVAR in the HOME directory so that it is executable from any subdirectory.

To remove this program, simply type: `'CLVAR' PURGE`

SYSEVAL

The SYSEVAL instruction is used to execute objects found in the HP 48 memory. Haphazard use of this function could cause a loss of memory.

This function could be considered dangerous, and you may want to prohibit its use. All you need to do is create a program with the same name: 'SYSEVAL'. As it is not normally possible to create such a name, we will use the program CRNAME.

To prohibit the use of SYSEVAL, do the following:

- Enter the following suggested program:

  « "SYSEVAL Not Available!" DOERR »

- Then type: "SYSEVAL" CRNAME STO

It is best to store this false 'SYSEVAL' program in the HOME directory so that it is executable from any subdirectory.

To remove this program, type: 'SYSEVAL' PURGE

Once the false program is installed, it is possible to enter the global name 'SYSEVAL' normally (without the use of CRNAME).

CONTRAST

CONTRAST uses the programs PEEK and POKE to change the HP 48's screen contrast. It takes a binary integer between #0h and #1Fh from the stack. #0h gives the lightest contrast, (the screen appears to be off), and #1Fh gives the darkest contrast (the screen appears completely black). This allows access to a greater range of contrast values than do the conventional [ON]-[+] and [ON]-[−] methods, which offer values from #3h to #13h.

```
CONTRAST(# 7BF1h)
  «
    HEX # 101h OVER # Fh AND →STR 3 3 SUB "#"
    # 102h # 1h PEEK + STR→ # Eh AND 4 ROLL 16
    / # 1h AND OR →STR 3 3 SUB + POKE
  »
```

DISPON and DISPOFF

DISPON and DISPOFF are two programs that use PEEK and POKE to turn the HP 48 screen on and off. Note that DISPOFF disables the keyboard, so the two programs must always be used together (always call DISPON after having called DISPOFF). If you execute DISPOFF alone, there is no way to turn the screen back on other than with a system halt ([ON]-[C]).

```
DISPON(# 10B7h)
  «
    # 100h "#" OVER # 1h PEEK + STR→ # 8h OR
    →STR 3 3 SUB POKE
  »
```

```
DISPOFF(# 8EF6h)
  «
    # 100h "#" OVER # 1h PEEK + STR→ # 7h AND
    →STR 3 3 SUB POKE
  »
```

FAST

FAST is a program that will enable you to speed up HP 48 calculations more than 12%. This program turns off the screen, (using the programs DISPOFF and DISPON), which lightens the bus load slightly, enabling the HP 48 to execute a little faster.

As an argument, FAST takes either a program, the name of a program, or a list of commands. If any of these arguments require arguments themselves, they must already be present on the stack.

Example: To calculate the second derivative of 'COS(COS(X))':

```
« 'COS(COS(X))' 'X' ∂ 'X' ∂ » FAST
```

```
FAST (# 14A3h)
  «
    DISPOFF
    IFERR
      EVAL
    THEN
      DISPON ERRN DOERR
    END
    DISPON
  »
```

DISASM

This fascinating program is monstrous in size but extremely useful: it can disassemble any machine language program. DISASM is the main program; all the others are its subroutines. It takes two arguments:

- In stack level 2, a character string which contains the hexadecimal codes that you wish to disassemble.
- In stack level 1, the beginning address of the code—useful when disassembling ROM programs (for movable programs, as are all programs in this book, give the value #0h for this argument).

  For example, to disassemble the routine at address #067B9h:

```
#067B9h DUP #100h PEEK SWAP DISASM
```

The disassembled code is found in the variable 'SOL' when DISASM has finished. The programs SPC1 and SPC2 in this listing are identical. They calculate the number of spaces between columns of the output listing given by DISASM. To change the column spacing, change one or the other.

DISASM can disassemble only machine language; it does not recognize object prologs, for example. Note that DISASM may terminate with an error if it lacks proper arguments or encounters an invalid code (e.g. 10E).

```
DISASM(# 8DACh)
 «
   HEX 64 STWS 'ADR' STO 'Z' STO
   "          - START -"
   10 CHR + 'SOL' STO 1 'P' STO Z SIZE
   → S
   «
     DO
       P 'I' STO L READ 1 + GET EVAL + STOS
     UNTIL
       P S >
     END
     "          - END - " STOS
   »
 »
```

```
TAKE (# 7AFDh)
   « Z P DUP SUB »


READ (# 3949h)
   « "#" Z P DUP SUB + STR→ B→R »


INC (# C417h)
   « 1 'P' STO+ »


STOS (# 3095h)
   «
      10 CHR + DUP 1 DISP SOL SWAP + 'SOL' STO INC
   »


L (# EB37h)
   { A0 A1 A2 A3 A4 A5 A6 A7
      A1 A9 AA AB AC AC AC AC }


A0 (# A89Bh)
   «
      INC READ DUP
      IF
         14 ≠
      THEN
         { "RTNSXM" "RTN" "RTNSC" "RTNCC" "SETHEX"
         "SETDEC" "RSTK=C" "C=RSTK" "CLRST" "C=ST"
         "ST=C" "CSTex" "P=P+1" "P=P-1" 14 "RTI" } SWAP
         1 + GET CODE SWAP
      ELSE
         DROP INC READ INC READ
         → x y
         «
            y 8 < 38 CHR 33 CHR IFTE
            → z
            «
               y 8 MOD 2 * 1 + "ABBCCADCBACBACCD"
               → t u
               «
                  u t DUP SUB u t 1 + DUP SUB
                  → a b
```

```
                 «
                    CODE a "=" a z b + + + + SPC2 +
                    IF
                      x 15 ==
                    THEN
                      "A"
                    ELSE
                      x CH
                    END
                 »
               »
             »
           »
       END
     »
```

A1 (# 484Eh)

```
     «
       { N M } "18" READ →STR POS GET INC READ 1 + GET
       EVAL
     »
```

N (# 956Ch)

```
     { C0 C0 C0 C0 C4 C4 C6 C6
       C6 C9 C9 C9 C6 C9 C9 C9 }
```

C0 (# 6508h)

```
     «
       TAKE INC CODE "P" 3 ROLL + STR→ READ 1 + GET
     »
```

C6 (# F0DAh)

```
     «
       { "D0=D0+" "D1=D1+" "D0=D0-" "D1=D1-" } READ 5 -
       DUP 4 > 3 * - GET INC CODE SWAP SPC2 READ 1 +
       →STR +
     »
```

```
C9 (# 95A9h)
  «
    READ 8 - DUP
    IF
      3 >
    THEN
      4 - "D1=("
    ELSE
      "D0=("
    END
    { 2 4 5 } ROT GET SWAP OVER + ")" + SPC2
    SWAP 1 -
    → x
    «
      INC Z P DUP x + SUB REVERSE + P x + 'P' STO
      CODE SWAP
    »
  »
```

```
C4 (# D7A3h)
  «
    READ INC READ
    → x y
    «
      { "DAT0=A" "DAT1=A" "A=DAT0" "A=DAT1" "DAT0=C"
      "DAT1=C" "C=DAT0" "C=DAT1" } y 8 MOD 1 + GET
      SPC2
      IF
        x 4 ==
      THEN
        IF
          y 8 <
        THEN
          "A"
        ELSE
          "B"
        END
      ELSE
        INC READ
        → z
        «
          IF
            y 8 <
          THEN
```

```
          z CH
        ELSE
          READ 1 + →STR
        END
      »
    END
    +
  »
  CODE SWAP
»
```

P0 (# E419h)

```
{ "R0=A" "R1=A" "R2=A" "R3=A" "R4=A" 5 6 7 "R0=C"
  "R1=C" "R2=C" "R3=C" "R4=C" }
```

P1 (# 9F7h)

```
{ "A=R0" "A=R1" "A=R2" "A=R3" "A=R4" 5 6 7 "C=R0"
  "C=R1" "C=R2" "C=R3" "C=R4" }
```

P2 (# D1C7h)

```
{ "AR0ex" "AR1ex" "AR2ex" "AR3ex" "AR4ex" 5 6 7
  "CR0ex" "CR1ex" "CR2ex" "CR3ex" "CR4ex" }
```

P3 (# 7E1Bh)

```
{ "D0=A" "D1=A" "AD0ex" "AD1ex" "D0=C" "D1=C"
  "CD0ex" "CD1ex" "D0=AS" "D1=AS" "AD0XS" "AD1XS"
  "D0=CS" "D1=CS" "CD0XS" "CD1XS" }
```

A2 (# 856Ah)

```
« INC CODE "P=" SPC2 READ →STR + »
```

A31 (# 6DCAh)

```
«
  INC READ
  → x
  «
    SPC2 Z INC P DUP x + DUP 'P' STO SUB
    REVERSE +
  »
»
```

```
A7 (# 1C34h)
   «
      "GOSUB" "" 1 3
      START
         INC TAKE +
      NEXT
      # 1000h 4 SAUTREL CODE SWAP
   »

A3 (# DB24h)
   « "LCHEX " A31 CODE SWAP »

A4 (# A72Dh)
   «
      INC TAKE INC TAKE + DUP
      IF
         "00" ==
      THEN
         DROP "RTNC"
      ELSE
         DUP
         IF
            "20" ==
         THEN
            DROP "NOP3"
         ELSE
            "GOC" SWAP # 100h 1 SAUTREL
         END
      END
      CODE SWAP
   »

A5 (# 4081h)
   «
      INC TAKE INC TAKE + DUP
      IF
         "00" ==
      THEN
         DROP "RTNNC"
      ELSE
         "GONC" SWAP # 100h 1 SAUTREL
      END
      CODE SWAP
   »
```

```
A6 (# A19Ch)
  «
     Z INC P DUP 3 + SUB DUP
     IF
        1 3 SUB "300" ==
     THEN
        DROP "NOP4"
     ELSE
        DUP
        IF
           "4000" ==
        THEN
         DROP "NOP5" INC
        ELSE
         1 3 SUB "GOTO" SWAP # 1000h 1 SAUTREL
        END
     END
     INC INC CODE SWAP
  »


M (# CC5Ch)
  { B0 B1 B1 B3 B4 B4 B6 B6
     B6 B6 BA BA BC BC BC BC }


B1 (# 9732h)
  «
     "U" TAKE + STR→ INC READ 1 + GET EVAL CODE SWAP
  »

B3 (# FA87h)
  « B1 GOYES »


B4 (# 5589h)
  «
     { "ST=0" "ST=1" } READ 3 - GET INC SPC2 READ
     →STR + CODE SWAP
  »
```

```
B0 (# E5CDh)
  «
    0 U0 INC READ
    → x
    «
      x 1 + GET
      IF
        x 8 ==
      THEN
        DROP2 INC { 6 7 10 11 } READ POS V0 READ
        1 + GET EVAL
      ELSE
        IF
          { 15 13 12 } x POS
        THEN
          SPC2 INC TAKE +
        END
      END
      CODE SWAP
      IF
        ROT
      THEN
        GOYES
      END
    »
  »

B6 (# 390Bh)
  «
    { "?ST=0" "?ST≠0" "?P≠" "?P=" } READ 5 - GET
    INC SPC2 READ →STR + CODE SWAP GOYES
  »

U0 (# 560Fh)
  { "OUT=CS" "OUT=C" "A=IN" "C=IN" "UNCNFG" "CONFIG"
    "C=ID" "SHUTDN" 8 "C+P+1" "RESET" "BUSCC" "C=P"
    "P=C" "SREQ?" "CPex" }
```

```
BA (# 2958h)
  «
    READ INC READ
    → x y
    «
      CODE
      IF
        x 10 ==
      THEN
        A
      ELSE
        B
      END
      y 1 + GET SPC2 + "A" GOYES
    »
  »


BC (# 2CCCh)
  «
    { "GOLONG" 4 "GOVLNG" 5 "GOSUBL" 4 "GOSBVL" 5 }
    READ 2 * 23 - DUP 1 + SUB LIST→ DROP
    → a b
    «
      a Z P 1 + DUP b + 1 - SUB
      IF
        b 5 ==
      THEN
        SWAP SPC2 SWAP REVERSE +
      ELSE
        # 10000h 2 READ 14 == 4 * + SAUTREL
      END
      P b + 'P' STO CODE SWAP
    »
  »


V0 (# E524h)
  { "INTON" V01 V02 "BUSCB" V04 V04 V04 V04 V04
    V04 V04 V04 "PC=(A)" "BUSCD" "PC=(C)"
    "INTOFF" }
```

```
U18 (# 8795h)
  «
    READ 8 == INC READ INC READ 1 +
    → t f r
    «
      RA r GET
      IF
        t
      THEN
        DUP "=" SWAP + +
        IF
          r 8 <
        THEN
          "+"
        ELSE
          "-"
        END
        + INC READ 1 + +
      ELSE
        "SRB" +
      END
      f CHA
    »
  »

V00 (# 33A5h)
  { "ABIT=0" "ABIT=1" "?ABIT=0" "?ABIT=1" "CBIT=0"
    "CBIT=1" "?CBIT=0" "?CBIT=1" }

V01 (# 22D6h)
  « INC "RSI" »

V02 (# 2584h)
  « "LAHEX " A31 »

V04 (# C703h)
  « V00 READ 3 - GET SPC2 INC TAKE + »

U1 (# CFB0h)
  { "ASLC" "BSLC" "CSLC" "DSLC" "ASRC" "BSRC"
    "CSRC" "DSRC" U18 U18 U1A U1B "ASRB" "BSRB"
    "CSRB" "DSRB" }
```

U1A (# BF19h)
```
«
   INC READ INC READ INC READ 1 +
   → f x r
   «
      RN r GET
      IF
         r 8 <
      THEN
         "A"
      ELSE
         "C"
      END
      IF
         x 2 ==
      THEN
         SWAP "ex" + +
      ELSE
         IF
            x 1 ==
         THEN
            SWAP
         END
            "=" SWAP + +
      END
      f CHA
   »
»
```

V1B (# BA48h)
```
{ 0 1 "PC=A" "PC=C" "A=PC" "C=PC" "APCex"
   "CPCex" }
```

U1B (# CC94h)
```
« V1B INC READ 1 + GET SPC2 "A" + »
```

RN (# FC36h)
```
{ "R0" "R1" "R2" "R3" "R4" 5 6 7 "R0" "R1" "R2"
   "R3" "R4" 13 14 15 }
```

```
RA (# 8ACEh)
  { "A" "B" "C" "D" 4 5 6 7 "A" "B" "C" "D" 12
    13 14 15 }

U2 (# 5EDBh)
  { 0 "XM=0" "SB=0" 3 "SR=0" 5 6 7 "MP=0" 9 10
    11 12 13 14 "CLRHST" }

U3 (# EA2Ch)
  { 0 "?XM=0" "?SB=0" 3 "?SR=0" 5 6 7 "?MP=0" }

A9 (# 48ADh)
  « A B NORMAL GOYES »

AA (# 2CB0h)
  « C D NORMAL »

AB (# B467h)
  « E F NORMAL »

AC (# BF15h)
  «
    { C D E F } READ 11 - GET EVAL INC CODE
    SWAP READ 1 + GET SPC2 "A" +
  »

A (# DD35h)
  { "?A=B" "?B=C" "?C=A" "?D=C" "?A≠B" "?B≠C" "?C≠A"
    "?D≠C" "?A=0" "?B=0" "?C=0" "?D=0" "?A≠0" "?B≠0"
    "?C≠0" "?D≠0" }

B (# 32E9h)
  { "?A>B" "?B>C" "?C>A" "?D>C" "?A<B" "?B<C" "?C<A"
    "?D<C" "?A≥B" "?B≥C" "?C≥A" "?D≥C" "?A≤B" "?B≤C"
    "?C≤A" "?D≤C" }
```

```
C (# 50AAh)
  { "A=A+B" "B=B+C" "C=C+A" "D=D+C" "A=A+A" "B=B+B"
    "C=C+C" "D=D+D" "B=B+A" "C=C+B" "A=A+C" "C=C+D"
    "A=A-1" "B=B-1" "C=C-1" "D=D-1" }

D (# 9930h)
  { "A=0" "B=0" "C=0" "D=0" "A=B" "B=C" "C=A" "D=C"
    "B=A" "C=B" "A=C" "C=D" "ABex" "CBex" "CAex"
    "CDex" }

E (# C345h)
  { "A=A-B" "B=B-C" "C=C-A" "D=D-C" "A=A+1" "B=B+1"
    "C=C+1" "D=D+1" "B=B-A" "C=C-B" "A=A-C" "C=C-D"
    "A=B-A" "B=C-B" "C=A-C" "D=C-D" }

F (# 7B66h)
  { "ASL" "BSL" "CSL" "DSL" "ASR" "BSR" "CSR" "DSR"
    "A=-A" "B=-B" "C=-C" "D=-D" "A=-A-1" "B=-B-1"
    "C=-C-1" "D=-D-1" }

SPC (# EA19h)
  "       "          (7 spaces)

SPC1 (# DF86h)
  « SPC 1 7 4 PICK SIZE - SUB + " " + »

SPC2 (# DF86h)
  « SPC 1 7 4 PICK SIZE - SUB + " " + »

ADRSTR (# 1EF0h)
  « # 100000h + →STR 4 8 SUB »

CODE (# A7D6h)
  «
    ADR I 1 - + ADRSTR " " + Z I P SUB SPC1 +
  »
```

```
SAUTREL(# D63Eh)
  «
    → a b c
    «
      SPC2 ADR I + 1 - c + "#" a REVERSE +
      OBJ→ DUP
      IF
        b 2 / <
      THEN
        +
      ELSE
        b SWAP - -
      END
      ADRSTR +
    »
  »

GOYES(# E103h)
  «
    + INC P 'I' STO TAKE INC TAKE +
    → a
    «
      10 CHR CODE
      IF
        a "00" ==
      THEN
        "RTNYES"
      ELSE
        "GOYES" a # 100h 0 SAUTREL
      END
      + +
    »
  »

NORMAL(# B551h)
  «
    → a b
    «
      INC READ INC READ
      → x y
      «
        CODE
        IF
          x 8 <
        THEN
```

```
         a
      ELSE
         b
      END
      y 1 + GET SPC2 x CH +
    »
  »
»
```

REVERSE (# B227h)

```
«
  → c
  «
   "" c SIZE 1
    FOR x
      c x DUP SUB + -1
    STEP
  »
»
```

CH (# 989Eh)

```
«
  → a
  «
    { "P" "WP" "XS" "X" "S" "M" "B" "W" } a 8
    MOD 1
    + GET
  »
»
```

CHA (# FDECh)

```
«
  → f
  «
    SPC2
    IF
      f 15 ==
    THEN
      "A"
    ELSE
      f CH
    END +
  »
»
```

Manipulating System Binaries

These programs convert between system binaries (SB) and other types of objects commonly used by the machine: binary integers (B), real numbers (R), and characters (C).

- The required arguments are not verified for these programs. You must be certain that you give the proper arguments if you would like to obtain the proper results (giving a bad argument will not damage the machine, just give unpredictable results).
- The character object is not normally accessible to the user . With the programs below, it can be easily generated. For example, to create the character #40h (A), you would type `#40h B→SB SB→C`. The corresponding character will appear as "Character" on the screen.

```
B→SB (# A92h)
  « # 5A03h SYSEVAL »

SB→B (# C4F4h)
  « # 59CCh SYSEVAL »

R→SB (# 41Ch)
  « # 18CEAh SYSEVAL »

SB→R (# F1E1h)
  « # 18DBFh SYSEVAL »

C→SB (# 2100h)
  « # 5A51h SYSEVAL »

SB→C (# 2756h)
  « # 5A75h SYSEVAL »
```

ROMRCL

This very short program can recall objects from ROM to the stack. Simply place the address of the object on the stack (as a binary integer), and execute `ROMRCL`.

First the program `B→SB` is used to convert the binary integer into a system binary, then the `#C621h SYSEVAL` is called to recall the object at the given address to the stack.

Notes:

- This program can recall objects in hidden ROM by duplicating them into RAM.
- Don't try random addresses.
- Don't use `ROMRCL` except for address in ROM.

```
ROMRCL (# B490h)
  « B→SB # C612h SYSEVAL »
```

A→STR and STR→A

A→STR transforms a binary integer address to a character string (written in reverse). STR→A does the opposite function. They are particularly useful when using PEEK and POKE to read and write addresses in memory. Each program uses the program REVERSE.

Examples:

#70000h A→STR returns "00007".

"0000F" STR→A returns # F0000h (in hexadecimal mode).

A→STR (# E4F3h)

```
«
  HEX # 100000h + # 1FFFFFh AND →STR REVERSE
  2 6 SUB
»
```

STR→A (# 9287h)

```
«
  "00000" + 1 5 SUB "h" SWAP + "#" + REVERSE
  STR→
»
```

BFREE

This program will determine the amount of free space left on a plug-in RAM card in BACKUP mode. It takes the port number as an argument, and returns the free space in bytes. BFREE uses PEEK and STR→A.

BFREE (# 6BE8h)

```
«
   → PORT
   «
      IF
         PORT 1 ≠ PORT 2 ≠ AND
      THEN
         # Ah DOERR
      END
      # 70421h PORT 11 * + → ADR
      «
         ADR # 1h PEEK STR→A → FLAGS
         «
            IF
               FLAGS # 8h AND # 0h ==
            THEN
               # Ah DOERR
            END
            IF
               FLAGS # 2h AND # 0h ≠
            THEN
               "CARD MERGED !" DOERR
            END
         »
         ADR 1 + # 5h PEEK STR→A # 100000h ADR
         6 + # 5h PEEK STR→A - + # 7044Dh PORT
         5 * + # 5h PEEK STR→A - B→R 2 /
      »
   »
»
```

SEARCH

Here are 3 programs for searching memory: ROMSEARCH, RAMSEARCH, and MODUSEARCH. These programs will search memory for a string of hexadecimal codes, and return the address(es) of any occurrences found.

- Use ROMSEARCH to search in ROM (including the hidden ROM). Addresses greater than #70000h (which are addresses of objects in the hidden ROM) should be used with ROMRCL to view the contents.
- Use RAMSEARCH to search in RAM (including merged plug-in cards).
- Use MODUSEARCH to search plug-in cards (HP 48SX only). This program takes one extra argument than the others: a real number that is the port number of the card you would like to search. After checking the port for the presence of a card, the search will be done. MODUSEARCH will search plug-in ROM cards as well as non-merged plug-in RAM cards.

Note: these three programs use the program SEARCH, as well as PEEK, HRPEEK (for ROMSEARCH) and STR→A (for RAMSEARCH and MODUSEARCH).

Examples:

- To find all character string objects in ROM: "C2A20" ROMSEARCH
- To do the same search in the plug-in card in port 2 (if the card is present): "C2A20" 2 MODUSEARCH

```
SEARCH(# EC79h)
  «
    → MOTIF AD FIN PRGM
    «
      # 100h DUP MOTIF SIZE + → LEN LENP
      «
        { }
        DO
          AD DUP 1 DISP LENP PRGM EVAL
          IF
            MOTIF POS AD OVER
          THEN
            + DUP 'AD' STO 1 - DUP
            IF
              FIN ≥
            THEN
              DROP
            ELSE
              DUP 2 DISP 1000 .07 BEEP +
            END
          ELSE
          + LEN + 'AD' STO
          END
        UNTIL
          AD FIN ≥
        END
      »
    »
  »
```

```
ROMSEARCH(# 5E4Eh)
  «
    → MOTIF
    «
      MOTIF # 0h # 70000h 'PEEK' SEARCH MOTIF
      # 70000h # 80000h 'HRPEEK' SEARCH +
    »
  »
```

```
RAMSEARCH(# 88ABh)
  «
    # 70000h # 70669h # 5h PEEK STR→A 'PEEK'
    SEARCH
  »
```

```
MODUSEARCH(# C06Dh)
  «
    → PORT
    «
      IF
        PORT 1 ≠ PORT 2 ≠ AND
      THEN
        # Ah DOERR
      END
      # 70421h PORT 11 * +
      → ADR
      «
        ADR #1h PEEK STR→A
        → FLAGS
        «
          IF
            FLAGS # 8h AND # 0h ==
          THEN
            # Ah DOERR
          END
          IF
            FLAGS # 2h AND # 0h ≠
          THEN
            "PORT MERGED-USE RAMS" DOERR
          END
        »
        ADR 1 + # 5h PEEK STR→A DUP # 100000h
        ADR 6 + # 5h PEEK STR→A - + 'PEEK' SEARCH
      »
    »
  »
```

CRC

This program calculates the cyclic redundancy control (CRC) used by the HP 48 to verify data in certain objects. The program takes a string of hexadecimal codes (like those accepted by GASS) and returns the corresponding checksum.

For example, "123456789ABCDEF0" CRC returns #A8ECh on the stack.

```
CRC (# 9D00h)
 «
   # 0h
    → S CRC.V
   «
     1 S SIZE
     FOR X
       S X X SUB NUM 48 - DUP 9 > 7 * - # 0h
       + CRC.V 16 / SWAP CRC.V XOR # Fh AND
       # 1081h * XOR 'CRC.V' STO
     NEXT
   CRC.V
 »
```

Here is a faster version written in machine language:

```
CRCLM (# D298h)
  D9D20 E4A20 A0000 00000 CB2A1 CCD20 CC000 8FB97
  60147 13416 91741 43131 17414 7D517 43450 000E1
  8A907 D014B 3103B 6A319 09EA9 03170 B6A14 67C50
  34F00 000EF 3DB80 82160 C7A6C 5AFCB 80821 40C7A
  6C5AF CB142 F4742 0DB14 41713 42000 06E8F 8F2D7
  60142 16480 8CD7F E0EF2 DFFC0 EF20E FB01D BBF18
  DBF1B 2130
```

CALC

CALC is a collection of programs that will perform arithmetic calculations with large integers. The HP 48 can already do arithmetic with integers, but only those in the range from 0 to 18446744073709551615. These programs can use integers that are as large as your memory will permit. As examples, they were used to calculate the factorial of 2000 (more than 5000 digits!), and the square root of 2, accurate to 500 decimal places.

These functions work with positive integers represented in string form. (For example, "1234567890" is the integer 1234567890). The functions:

- ADD to add two integers;
- SUBS to subtract two integers and return the absolute value;
- MULT to multiply two integers;
- BFACT to calculate the factorial of the integer given as an argument. It does this by making successive multiplications, and displays on the screen the current result as well as the number of multiplications left, so that the user can get an idea as to when it will be finished.
- POW will raise the integer in level 2 to the power in level 1 (just like the ^ function). As with BFACT, step numbers are displayed to show what work is left to do (0 will be displayed when it's done).
- E multiplies the integer in level 2 by 10 raised to the power in level 1.
- DIV divides the integer in level 2 by the integer in level 1, and returns the integer part.
- MODU is the modulo function. It returns the remainder of the integer in level 2 divided by the integer in level 1.
- SQR calculates the integer part of the square root of the argument given.

These programs all use subroutines, most of which are written in assembly. The commented source listings are first, then the hexadecimal codes.

DECODE.LM

This program converts an integer in a special format used by ADD.LM, SUB.LM, and MULT.LM into an integer in string form.

```
      CCD20     CON(5)    PROL_CODE        Code object
begin B6000     CON(5)    (end)-(begin)    Code Length
      8FB9760   GOSBVL    SAVE_REG         Backup regs.
      143       A=DAT1    A
      132       AD0ex                      D0=address of object
                                           in stack level 1
      164       D0=D0+    5
      3450000   LCHEX     #00005
      142       A=DAT0    A                Object length
      EA        A=A-C     A
      D8        B=A       A
      164       D0=D0+    5
      174       D1=D1+    5
      143       A=DAT1    A
      133       AD1ex                      D1=address of object
                                           in stack level 2
      174       D1=D1+    5
      147       C=DAT1    A
      133       AD1ex
      C2        C=C+A     A
      137       CD1ex
      3103      LCHEX     #30
l1    8A9       ?B=0      A                Done?
      61        GOYES     l2               Yes --> end
      1C1       D1=D1-    2
      15E0      C=DAT0    1                Read a digit
      15D1      DAT1=C    2
      160       D0=D0+    1
      CD        B=B-1     A                One digit less
      6AEF      GOTO l1
l2    8F2D760   GOSBVL    LOAD_REG         Restore regs.
      142       A=DAT0    A                Return to RPL
      164       D0=D0+    5
      808C      PC=(A)
end
```

ENCODE.LM

This is the inverse function of DECODE.LM. It will convert an integer in string form into an integer in a special format.

```
     CCD20    CON(5)  PROL_CODE      Code object
begin76000    CON(5)  (end)-(begin)  Code length
     8FB9760  GOSBVL  SAVE_REG       Backup regs.
     143      A=DAT1  A
     132      AD0ex                  D0=Address of object
                                        in stack level 1
     164      D0=D0+  5
     3450000  LCHEX   #00005
     142      A=DAT0  A              Object length
     EA       A=A-C   A
     D8       B=A     A
     164      D0=D0+  5
     174      D1=D1+  5
     143      A=DAT1  A              D1=Address of object
                                        in stack level 2
     133      AD1ex
     174      D1=D1+  5
     147      C=DAT1  A
     133      AD1ex
     C2       C=C+A   A
     137      CD1ex
l1   8A9      ?B=0    A              Done?
     61       GOYES   l2             Yes --> end
     1C1      D1=D1-  2
     15B0     A=DAT1  1              Read 1 digit
     1500     DAT0=A  P
     160      D0=D0+  1
     CD       B=B-1   A              One digit less
     6AEF     GOTO l  1              Loop
l2   8F2D760  GOSBVL  LOAD_REG       Restore regs.
     142      A=DAT0  A              Return to RPL
     164      D0=D0+  5
     808C     PC=(A)
end
```

FORMAT.LM

This program will remove any leading zeros from an integer (convert "00123" to "123", for example).

```
     CCD20    CON(5)   PROL_CODE        Code object
begin5E000    CON(5)   (end)-(begin)    Code length
     8FB9760  GOSBVL   SAVE_REG         Backup regs.
     143      A=DAT1   A
     130      D0=A                      D0=Address of object
                                           in stack level 1
     169      D0=D0+   10
     174      D1=D1+   5
     143      A=DAT1   A
     131      D1=A                      D1=Address of object
                                           in stack level 2
     174      D1=D1+   5
     143      A=DAT1   A                Object length
     818F84   A=A-5    A
     172      D1=D1+   3
     D3       D=0      A                Number of zeroes
                                           to remove
l1   171      D1=D1+   2
     E7       D=D+1    A
     818F81   A=A-2    A
     8A8      ?A=0     A                Done?
     B0       GOYES    l2               Yes --> end
     1570     C=DAT1   P
     90A      ?C=0     P                A zero?
     9E       GOYES    l1               Yes --> loop
l2   DB       C=D  A
     144      DAT0=C   A                write the number of
                                           zeros to remove
     8F2D760  GOSBVL   LOAD_REG         Restore regs.
     142      A=DAT0   A                Return to RPL
     164      D0=D0+   5
     808C     PC=(A)
end
```

ZERO.LM

This program sets the integer given as an argument to zero, in the special integer format.

```
     CCD20    CON(5)  PROL_CODE      Code object
begin54000    CON(5)  (end)-(begin)  Code length
     8FB9760  GOSBVL  SAVE_REG       Backup regs.
     143      A=DAT1  A
     131      D1=A                   D1=Address of object
                                        in stack level 1
     174      D1=D1+  5
     143      A=DAT1  A
     174      D1=D1+  5
     C4       A=A+A   A
     F4       ASR     A              A=number of 8-digit
                                     blocks for this object
     AF2      C=0     W
l1   8A8      ?A=0    A              Done?
     F0       GOYES   l2             Yes --> end
     15D7     DAT1=C  8              Set to zero
     177      D1=D1+  8
     CC       A=A-1   A
     61FF     GOTO    l1             Loop
l2   8F2D760  GOSBVL  LOAD_REG       Restore regs.
     142      A=DAT0  A              Return to RPL.
     164      D0=D0+  5
     808C     PC=(A)
end
```

ADD.LM

This program will add two integers. It works with blocks of 8 digits.

```
     CCD20    CON(5)   PROL_CODE        Code object
begin57000    CON(5)   (end)-(begin)    Code length
     8FB9760  GOSBVL   SAVE_REG         Backup regs.
     143      A=DAT1   A
     130      D0=A
     169      D0=D0+   10               D0=Address of object
                                          in stack level 1
     174      D1=D1+   5
     143      A=DAT1   A
     131      D1=A                      D1=Address of object
                                          in stack level 2
     174      D1=D1+   5
     147      C=DAT1   A
     C6       C=C+C    A
     F6       CSR      A
     D7       D=C      A                D= # of blocks for obj.
     174      D1=D1+   5
     AF0      A=0      W
     20       P=       0                Carry to zero
l1   8AB      ?D=0     A                Done?
     F2       GOYES    l2               Yes --> end
     AF2      C=0      W
     80F0     CPex     0                Carry
     05       SETDEC                    Decimal mode
     15A7     A=DAT0   8                Read first block
     A72      C=C+A    W                Add to carry
     15B7     A=DAT1   8                Read second block
     A72      C=C+A    W                Add
     04       SETHEX                    Hexadecimal mode
     15D7     DAT1=C   8                Read result
     167      D0=D0+   8                Next blocks
     177      D1=D1+   8
     CF       D=D-1    A                One block less
     80D8     P=C      8                Carry --> P
     61DF     GOTO     l1               Loop
l2   8F2D760  GOSBVL   LOAD_REG         Restore regs.
     142      A=DAT0   A                Return to RPL
     164      D0=D0+   5
     808C     PC=(A)
end
```

SUB.LM

This program will subtract two integers. It works with blocks of 8 digits.

```
     CCD20    CON(5)  PROL_CODE        Code object
begin67000    CON(5)  (end)-(begin)    Code length
     8FB9760  GOSBVL  SAVE_REG         Backup regs.
     143      A=DAT1  A
     130      D0=A                     D0=Address of object
     169      D0=D0+  10                 in stack level 1
     174      D1=D1+  5
     143      A=DAT1  A
     131      D1=A                     D1=Address of object
     174      D1=D1+  5                  in stack level 2
     147      C=DAT1  A
     C6       C=C+C   A
     F6       CSR     A
     D7       D=C     A                D= # of blocks in obj.
     174      D1=D1+  5
     AF0      A=0     W                No carry
l1   8AB      ?D=0    A                Done?
     23       GOYES   l2               Yes --> end
     AF2      C=0     W
     15E7     C=DAT0  8                Read 1 block
     05       SETDEC                   Decimal mode
     A7A      A=A+C   W                Add to carry
     15F7     C=DAT1  8                Block to subtract
     B72      C=C-A   W                Subtraction
     04       SETHEX                   Hexadecimal mode
     15D7     DAT1=C  8                Write result
     177      D1=D1+  8
     167      D0=D0+  8
     CF       D=D-1   A                One block less
     AF0      A=0     W
     94A      ?C=0    S                Carry?
     4D       GOYES   l1               No --> loop
     B64      A=A+1   B                Save the carry
     6ECF     GOTO    l1               Loop
l2   8F2D760  GOSBVL  LOAD_REG         Restore regs.
     142      A=DAT0  A                Return to RPL
     164      D0=D0+  5
     808C     PC=(A)
end
```

MULT.LM

This program will multiply two integers. It does this calculation much like you would do it by hand on paper by working with one digit at a time.

```
      CCD20    CON(5)   PROL_CODE        Code object
beginC1100     CON(5)   (end)-(begin)    Code length
      8FB9760  GOSBVL   SAVE_REG         Backup regs.
      143      A=DAT1   A
      818F09   A=A+10   A
      101      R1=A                      R1=address of con-
                                            tents of level-1
                                            object (the result)
      174      D1=D1+   5
      143      A=DAT1   A
      133      AD1ex                     D1=Address of object
                                            in stack level 2
      174      D1=D1+   5
      AF2      C=0      W
      147      C=DAT1   A
      818FA4   C=C-5    A
      BF2      CSL      W
      BF2      CSL      W
      BF2      CSL      W
      AD7      D=C      M                Number of blocks of
                                            integer in level 2
      174      D1=D1+   5
      133      AD1ex
      103      R3=A                      R3=address of con-
                                            tents of level-2 obj.
      174      D1=D1+   5
      143      A=DAT1   A
      131      D1=A                      D1=address of object
                                            in stack level 3
      174      D1=D1+   5
      AF2      C=0      W
      147      C=DAT1   A
      818FA4   C=C-5    A
      BF2      CSL      W
      BF2      CSL      W
      BF2      CSL      W
      174      D1=D1+   5
      133      AD1ex
```

```
        102     R2=A               R2=address of object-
                                       3 contents
l1      95F     ?D#0       M       More work?
        60      GOYES      l2      Yes --> continue
        6690    GOTO       l7      No --> stop
l2      113     A=R3
        131     D1=A
        AE2     C=0        B
        15F0    C=DAT1     1       Read a digit
        AE7     D=C        B
        170     D1=D1+     1
        133     AD1ex
        103     R3=A
        A5F     D=D-1      M       One less digit
        112     A=R2
        131     D1=A
        ADD     CBex       M
        AD9     C=B        M
        111     A=R1
        130     D0=A
        E4      A=A+1      A
        101     R1=A
        96B     ?D=0       B       Mult by zero?
        1C      GOYES      l1      Yes --> done
        AE2     C=0        B
l3      95D     ?B#0       M       Again?
        A0      GOYES      l4      Yes
        15C0    DAT0=C     1       No --> write final carry
        62BF    GOTO       l1      And loop
l4      06      RSTK=C             Backup C
        15B0    A=DAT1     1       Read a digit
        AEB     C=D        B
        05      SETDEC             Decimal mode
        AE1     B=0        B
l5      822     SB=0               ;
        81900   ASRB       P       ;
        832     ?SB=0              ;
        50      GOYES      l6      ;  Multiplication
        A61     B=B+C      B       ;
l6      A66     C=C+C      B       ;
        90C     ?A#0       P       ;
        AE      GOYES      l5      ;
        07      C=RSTK
        A69     C=C+B      B       Add the carry
        AE0     A=0        B
```

```
      15A0     A=DAT0   1
      A62      C=C+A    B                 Add to existing
      15C0     DAT0=C   1                 Write result
      04       SETHEX                     Hexadecimal mode
      160      D0=D0+   1
      170      D1=D1+   1
      A5D      B=B-1    M
      BE6      CSR      B                 Update carry
      6BAF     GOTO     13                Loop
l7    8F2D760  GOSBVL   LOAD_REG          Restore regs.
      142      A=DAT0   A                 Return to RPL
      164      D0=D0+   5
      808C     PC=(A)
end
```

DIV.LM

This program divides two integers and returns the integer part of the result.

```
      CCD20    CON(5)   PROL_CODE          Code object
begin 76100    CON(5)   (end)-(begin)      Code length
      8FB9760  GOSBVL   SAVE_REG           Backup regs.
      143      A=DAT1   A
      130      D0=A                        D0=Address of object
                                             in stack level 1
      164      D0=D0+   5
      142      A=DAT0   A
      818F84   A=A-5    A
      819F0    ASRB     A
      103      R3=A                        R3= # of digits
      164      D0=D0+   5
      132      AD0ex
      102      R2=A                        R2=address of con-
                                             tents of level-1 obj.
      174      D1=D1+   5                  Next object
      143      A=DAT1   A                  A=Address of object
                                             in stack level 2
      818F07   A=A+8    A
      D8       B=A      A
      174      D1=D1+   5                  Next object
      143      A=DAT1   A                  A=Address of object
                                             in stack level 3
      130      D0=A
      167      D0=D0+   8
      174      D1=D1+   5                  Next object
      143      A=DAT1   A                  A=Address of object
                                             in stack level 4
      818F04   A=A+5    A
      131      D1=A
      147      C=DAT1   A                  object 4 length
      CA       A=A+C    A
      818F81   A=A-2    A
      100      R0=A
      818FA4   C=C-5    A
      819F2    CSRB     A                  # of digits in object 4
      109      R1=C
      AC3      D=0      S
l1    113      A=R3
```

```
      8AC    ?A#0     A      Again?
      60     GOYES    l2
      61B0   GOTO     l9     No --> end
l2    CC     A=A-1    A      One less digit
      103    R3=A
      AC2    C=0      S
l3    111    A=R1            ;
      DE     CAex     A      ;
      D7     D=C      A      ;
      DE     CAex     A      ;
      136    CD0ex           ;
      C2     C=C+A    A      ;
      C2     C=C+A    A      ;
      C8     B=B+A    A      ; Initializations
      C8     B=B+A    A      ;
      DD     CBex     A      ;
      136    CD0ex           ;
      B47    D=D+1    S      ;
      110    A=R0            ;
      131    D1=A            ;
      AE2    C=0      B      No carry
l4    8AF    ?D#0     A      Again?
      60     GOYES    l5
      6740   GOTO     l7     No --> next
l5    CF     D=D-1    A
      05     SETDEC          Decimal mode
      AE0    A=0      B
      15B0   A=DAT1   1      Read 1 digit
      A62    C=C+A    B      Add to carry
      15A0   A=DAT0   1
      EE     C=A-C    A      Subtract
      04     SETHEX
      DC     ABex     A
      132    AD0ex
      15C0   DAT0=C   1      Re-write
      181    D0=D0-   2
      132    AD0ex
      DC     ABex     A
      181    D0=D0-   2      One less digit
      1C1    D1=D1-   2
      A82    C=0      P
      96A    ?C=0     B      Carry?
      A0     GOYES    l6
      3110   LCHEX    01     Yes --> carry
      6DBF   GOTO     l4
```

```
l6    AE2      C=0      B          No carry
      66BF     GOTO     l4
l7    96E      ?C#0     B          Carry at end
      90       GOYES    l8         Yes --> stop
      B46      C=C+1    S          Increment quotient
      658F     GOTO     l3         Loop
l8    161      D0=D0+   2
      136      CD0ex
      DD       CBex     A
      136      CD0ex
      161      D0=D0+   2
      B47      D=D+1    S
      112      A=R2
      131      D1=A
      1554     DAT1=C   S          Write quotient
      171      D1=D1+   2
      133      AD1ex
      102      R2=A
      694F     GOTO     l1         Loop
l9    8F2D760  GOSBVL   LOAD_REG   Restore regs.
      822      SB=0
      81943    DSRB     S
      832      ?SB=0               Need to change order
                                      of stack objects?
      A1       GOYES    l10        No --> end
      174      D1=D1+   5          ;
      143      A=DAT1   A          ;
      174      D1=D1+   5          ;
      147      C=DAT1   A          ;Exchange objects
      141      DAT1=A   A          ;in level 2 and
      1C4      D1=D1-   5          ;level 3
      145      DAT1=C   A          ;
      1C4      D1=D1-   5          ;
l10   142      A=DAT0   A          Return to RPL
      164      D0=D0+   5
      808C     PC=(A)
end
```

DECODE.LM(# D620h)
```
CCD20 B6000 8FB97 60143 13216 43450 00014 2EAD8
16417 41431 33174 14713 3C213 73103 8A961 1C115
E015D 1160C D6AEF 8F2D7 60142 16480 8C
```

ENCODE.LM(# B0A9h)
```
CCD20 76000 8FB97 60143 13216 43450 00014 2EAD8
16417 41431 33174 14713 3C213 78A96 11C11 5B015
00160 CD6AE F8F2D 76014 21648 08C
```

FORMAT.LM(# 3371h)
```
CCD20 E5000 8FB97 60143 13016 91741 43131 17414
3818F 84172 D3171 E7818 F818A 8B015 7090A 9EDB1
448F2 D7601 42164 808C
```

ZERO.LM(# 69AAh)
```
CCD20 54000 8FB97 60143 13117 41431 74C4F 4AF28
A8F01 5D717 7CC61 FF8F2 D7601 42164 808C
```

ADD.LM(# E74Ch)
```
CCD20 57000 8FB97 60143 13016 91741 43131 17414
7C6F6 D7174 AF020 8ABF2 AF280 F0051 5A7A7 215B7
A7204 15D71 67177 CF80D 861DF 8F2D7 60142 16480
8C
```

SUB.LM(# C14h)
```
CCD20 67000 8FB97 60143 13016 91741 43131 17414
7C6F6 D7174 AF08A B23AF 215E7 05A7A 15F7B 72041
5D717 7167C FAF09 4A4DB 646EC F8F2D 76014 21648
08C
```

MULT.LM(# ACDBh)
```
CCD20 C1100 8FB97 60143 818F0 91011 74143 13317
4AF21 47818 FA4BF 2BF2B F2AD7 17413 31031 74143
13117 4AF21 47818 FA4BF 2BF2B F2174 13310 295F6
06690 11313 1AE21 5F0AE 71701 33103 A5F11 2131A
DDAD9 11113 0E410 196B1 CAE29 5DA01 5C062 BF061
5B0AE B05AE 18228 19008 3250A 61A66 90CAE 07A69
AE015 A0A62 15C00 41601 70A5D BE66B AF8F2 D7601
42164 808C
```

```
DIV.LM (# AD61h)
  CCD20 76100 8FB97 60143 13016 41428 18F84 819F0
  10316 41321 02174 14381 8F07D 81741 43130 16717
  41438 18F04 13114 7CA81 8F811 00818 FA481 9F210
  9AC31 138AC 6061B 0CC10 3AC21 11DED 7DE13 6C2C2
  C8C8D D136B 47110 131AE 28AF6 06740 CF05A E015B
  0A621 5A0EE 04DC1 3215C 01811 32DC1 811C1 A8296
  AA031 106DB FAE26 6BF96 E90B4 6658F 16113 6DD13
  6161B 47112 13115 54171 13310 2694F 8F2D7 60822
  81943 832A1 17414 31741 47141 1C414 51C41 42164
  808C
```

```
DIV.C (# B5C2h)
  «
    FORMAT "0" SWAP + SWAP FORMAT "0" SWAP +
    IF
      OVER "00" ==
    THEN
      DROP2 # 305h DOERR
    ELSE
      DUP NEWOB DUP 1 OVER SIZE 6 PICK SIZE - 1 +
      SUB DIV.LM SWAP ROT DROP DUP SIZE DUP 5 ROLL
      SIZE - 1 + SWAP SUB
    END
  »
```

```
MULT.C (# 7E7Ch)
  « DUP2 + ZERO.LM MULT.LM 3 ROLLD DROP2 »
```

```
PREPARE (# 18D6h)
  «
    FORMAT SWAP FORMAT → N1 N2
    «
      IF
        N1 SIZE N2 SIZE DUP2 >
      THEN
        DROP2 N2 N1
      ELSE
        IF
          <
        THEN
          N1 N2
```

```
        ELSE
          N1 N2
          IF
            DUP2 >
          THEN
            SWAP
          END
        END
      END
      ENCODE SWAP ENCODE DUP2 SIZE SWAP SIZE SWAP
      - 0 CHR
      WHILE
        DUP2 SIZE >
      REPEAT
        DUP +
      END
      1 ROT SUB +
    »
  »
```

```
DECODE (# A04Dh)
  « DUP DUP + SWAP DECODE.LM DROP FORMAT »
```

```
ENCODE (# 19ADh)
  «
    "0000000" SWAP + DUP SIZE 8 MOD 1 + OVER SIZE
    SUB DUP 1 OVER SIZE 2 / SUB ENCODE.LM SWAP
    DROP
  »
```

```
FORMAT (# E1B2h)
  «
    "0" SWAP + # FFFFFh NEWOB FORMAT.LM B→R OVER
    SIZE SUB
  »
```

MODU (# FB90h)

```
   «
      IF
         FORMAT DUP "0" ==
      THEN
         DROP
      ELSE
         DIV.C SWAP DROP FORMAT
      END
   »
```

DIV (# 600Ah)

```
   « DIV.C DROP FORMAT »
```

E (# 5A9Eh)

```
   «
      →STR STR→ DUP
      → N
      «
         "0"
         WHILE
            N DUP 2 / IP 'N' STO
         REPEAT
            DUP +
         END
         1 ROT SUB +
      »
   »
```

POW (# D4DBh)

```
   «
      →STR STR→
      → N
      «
         ENCODE 1 ENCODE
         WHILE
            N DUP 1 DISP 0 ≠
         REPEAT
            IF
               N 2 / DUP IP 'N' STO FP
            THEN
               OVER MULT.C
```

```
        END
        SWAP DUP MULT.C SWAP
      END
      SWAP DROP DECODE
    »
  »
```

```
SQR (# C265h)
  «
    "00" + FORMAT DUP 1 OVER SIZE 2 / SUB
    → A X
    «
      DO
        X A OVER DIV ADD 2 DIV
      UNTIL
        X OVER 'X' STO ==
      END
      X 1 OVER SIZE 1 - SUB
    »
  »
```

```
BFACT (# 23E5h)
  «
    →STR STR→ DUP 2 DISP 1 ENCODE 1 ROT
    FOR X
      X DUP 1 DISP ENCODE MULT.C
    NEXT
    DECODE
  »
```

```
MULT (# EC5Fh)
  « ENCODE SWAP ENCODE MULT.C DECODE »
```

```
SUBS (# 204Fh)
  « PREPARE SUB.LM DROP DECODE »
```

```
ADD (# 701Ch)
  «
    PREPARE 0 CHR DUP + DUP + ROT OVER + 3 ROLLD +
    ADD.LM DROP DECODE
  »
```

Factorial 2000

This result was obtained using the programs in CALC, listed previously.

```
331,627,509,245,063,324,117,539,338,057,632,403,828,111,720,810,578,039,
457,193,543,706,038,077,905,600,822,400,273,230,859,732,592,255,402,352,
941,225,834,109,258,084,817,415,293,796,131,386,633,526,343,688,905,634,
058,556,163,940,605,117,252,571,870,647,856,393,544,045,405,243,957,467,
037,674,108,722,970,434,684,158,343,752,431,580,877,533,645,127,487,995,
436,859,247,408,032,408,946,561,507,233,250,652,797,655,757,179,671,536,
718,689,359,056,112,815,871,601,717,232,657,156,110,004,214,012,420,433,
842,573,712,700,175,883,547,796,899,921,283,528,996,665,853,405,579,854,
903,657,366,350,133,386,550,401,172,012,152,635,488,038,268,152,152,246,
920,995,206,031,564,418,565,480,675,946,497,051,552,288,205,234,899,995,
726,450,814,065,536,678,969,532,101,467,622,671,332,026,831,552,205,194,
494,461,618,239,275,204,026,529,722,631,502,574,752,048,296,064,750,927,
394,165,856,283,531,779,574,482,876,314,596,450,373,991,327,334,177,263,
608,852,490,093,506,621,610,144,459,709,412,707,821,313,732,563,831,572,
302,019,949,914,958,316,470,942,774,473,870,327,985,549,674,298,608,839,
376,326,824,152,478,834,387,469,595,829,257,740,574,539,837,501,585,815,
468,136,294,217,949,972,399,813,599,481,016,556,563,876,034,227,312,912,
250,384,709,872,909,626,622,461,971,076,605,931,550,201,895,135,583,165,
357,871,492,290,916,779,049,702,247,094,611,937,607,785,165,110,684,432,
255,905,648,736,266,530,377,384,650,390,788,049,524,600,712,549,402,614,
566,072,254,136,302,754,913,671,583,406,097,831,074,945,282,217,490,781,
347,709,693,241,556,111,339,828,051,358,600,690,594,619,965,257,310,741,
177,081,519,922,564,516,778,571,458,056,602,185,654,760,952,377,463,016,
679,422,488,444,485,798,349,801,548,032,620,829,890,965,857,381,751,888,
619,376,692,828,279,888,453,584,639,896,594,213,952,984,465,291,092,009,
103,710,046,149,449,915,828,588,050,761,867,924,946,385,180,879,874,512,
891,408,019,340,074,625,920,057,098,729,578,599,643,650,655,895,612,410,
231,018,690,556,060,308,783,629,110,505,601,245,908,998,383,410,799,367,
902,052,076,858,669,183,477,906,558,544,700,148,692,656,924,631,933,337,
612,428,097,420,067,172,846,361,939,249,698,628,468,719,993,450,393,889,
367,270,487,127,172,734,561,700,354,867,477,509,102,955,523,953,547,941,
107,421,913,301,356,819,541,091,941,462,766,417,542,161,587,625,262,858,
089,801,222,443,890,248,677,182,054,959,415,751,991,701,271,767,571,787,
495,861,619,665,931,878,855,141,835,782,092,601,482,071,777,331,735,396,
034,304,969,082,070,589,958,701,381,980,813,035,590,160,762,908,388,574,
561,288,217,698,136,182,483,576,739,218,303,118,414,719,133,986,892,842,
344,000,779,246,691,209,766,731,651,433,494,437,473,235,636,572,048,844,
478,331,854,941,693,030,124,531,676,232,745,367,879,322,847,473,824,485,
092,283,139,952,509,732,505,979,127,031,047,683,601,481,191,102,229,253,
372,697,693,823,670,057,565,612,400,290,576,043,852,852,902,937,606,479,
533,458,179,666,123,839,605,262,549,107,186,663,869,354,766,108,455,046,
198,102,084,050,635,827,676,526,589,492,393,249,519,685,954,171,672,419,
329,530,683,673,495,544,004,586,359,838,161,043,059,449,826,627,530,605,
423,580,755,894,108,278,880,427,825,951,089,880,635,410,567,917,950,974,
017,780,688,782,869,810,219,010,900,148,352,061,688,883,720,250,310,665,
922,068,601,483,649,830,532,782,088,263,536,558,043,605,686,781,284,169,
217,133,047,141,176,312,175,895,777,122,637,584,753,123,517,230,990,549,
829,210,134,687,304,205,898,014,418,063,875,382,664,169,897,704,237,759,
406,280,877,253,702,265,426,530,580,862,379,301,422,675,821,187,143,502,
918,637,636,340,300,173,251,818,262,076,039,747,369,595,202,642,632,364,
145,446,851,113,427,202,150,458,383,851,010,136,941,313,034,856,221,916,
```

631,623,892,632,765,815,355,011,276,307,825,059,969,158,824,533,457,435,
437,863,683,173,730,673,296,589,355,199,694,458,236,873,508,830,278,657,
700,879,749,889,992,343,555,566,240,682,834,763,784,685,183,844,973,648,
873,952,475,103,224,222,110,561,201,295,829,657,191,368,108,693,825,475,
764,118,886,879,346,725,191,246,192,151,144,738,836,269,591,643,672,490,
071,653,428,228,152,661,247,800,463,922,544,945,170,363,723,627,940,757,
784,542,091,048,305,461,656,190,622,174,286,981,602,973,324,046,520,201,
992,813,854,882,681,951,007,282,869,701,070,737,500,927,666,487,502,174,
775,372,742,351,508,748,246,720,274,170,031,581,122,805,896,178,122,160,
747,437,947,510,950,620,938,556,674,581,252,518,376,682,157,712,807,861,
499,255,876,132,352,950,422,346,387,878,954,850,885,764,466,136,290,394,
127,665,978,044,202,092,281,337,987,115,900,896,264,878,942,413,210,454,
925,003,566,670,632,909,441,579,372,986,743,421,470,507,213,588,932,019,
580,723,064,781,498,429,522,595,589,012,754,823,971,773,325,722,910,325,
760,929,790,733,299,545,056,388,362,640,474,650,245,080,809,469,116,072,
632,087,494,143,973,000,704,111,418,595,530,278,827,357,654,819,182,002,
449,697,761,111,346,318,195,282,761,590,964,189,790,958,117,338,627,206,
088,910,432,945,244,978,535,147,014,112,442,143,055,486,089,639,578,378,
347,325,323,595,763,291,438,925,288,393,986,256,273,242,862,775,563,140,
463,830,389,168,421,633,113,445,636,309,571,965,978,466,338,551,492,316,
196,335,675,355,138,403,425,804,162,919,837,822,266,909,521,770,153,175,
338,730,284,610,841,886,554,138,329,171,951,332,117,895,728,541,662,084,
823,682,817,932,512,931,237,521,541,926,970,269,703,299,477,643,823,386,
483,008,871,530,373,405,666,383,868,294,088,487,730,721,762,268,849,023,
084,934,661,194,260,180,272,613,802,108,005,078,215,741,006,054,848,201,
347,859,578,102,770,707,780,655,512,772,540,501,674,332,396,066,253,216,
415,004,808,772,403,047,611,929,032,210,154,385,353,138,685,538,486,425,
570,790,795,341,176,519,571,188,683,739,880,683,895,792,743,749,683,498,
142,923,292,196,309,777,090,143,936,843,655,333,359,307,820,181,312,993,
455,024,206,044,563,340,578,606,962,471,961,505,603,394,899,523,321,800,
434,359,967,256,623,927,196,435,402,872,055,475,012,079,854,331,970,674,
797,313,126,813,523,653,744,085,662,263,206,768,837,585,132,782,896,252,
333,284,341,812,977,624,697,079,543,436,003,492,343,159,239,674,763,638,
912,115,285,406,657,783,646,213,911,247,447,051,255,226,342,701,239,527,
018,127,045,491,648,045,932,248,108,858,674,600,952,306,793,175,967,755,
581,011,679,940,005,249,806,303,763,141,344,412,269,037,034,987,355,799,
916,009,259,248,075,052,485,541,568,266,281,760,815,446,308,305,406,677,
412,630,124,441,864,204,108,373,119,093,130,001,154,470,560,277,773,724,
378,067,188,899,770,851,056,727,276,781,247,198,832,857,695,844,217,588,
895,160,467,868,204,810,010,047,816,462,358,220,838,532,488,134,270,834,
079,868,486,632,162,720,208,823,308,727,819,085,378,845,469,131,556,021,
728,873,121,907,393,965,209,260,229,101,477,527,080,930,865,364,979,858,
554,010,577,450,279,289,814,603,688,431,821,508,637,246,216,967,872,282,
169,347,370,599,286,277,112,447,690,920,902,988,320,166,830,170,273,420,
259,765,671,709,863,311,216,349,502,171,264,426,827,119,650,264,054,228,
231,759,630,874,475,301,847,194,095,524,263,411,498,469,508,073,390,080,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000.

PI

Calculating π has always been a fascinating problem for mathematicians. Today, with computers, it is possible to calculate π accurately to millions of decimal places. Using the CALC programs, we will also make this calculation. However, because of the limited RAM, we can only calculate a few thousand decimal places.

There is a well known formula:

$$\frac{\pi}{4} = A\tan\left(\frac{1}{2}\right) + A\tan\left(\frac{1}{5}\right) + A\tan\left(\frac{1}{8}\right)$$

And we know that Atan can be calculated by:

$$A\tan(x) + \sum_{n=0}^{\infty} (-1)^n \cdot \frac{x^{2n+1}}{2n+1}$$

which converges faster as x gets smaller.

We have, therefore:

$$\pi = 4 \cdot \left(\sum_{n=0}^{\infty} (-1)^n \cdot \frac{\left(\frac{1}{2}\right)^{2n+1} + \left(\frac{1}{5}\right)^{2n+1} + \left(\frac{1}{8}\right)^{2n+1}}{2n+1} \right)$$

As CALC can only manage positive integers, we must multiply everything by a power of 10, and keep track of the sign manually. The program PI makes this calculation. It takes a real number from the stack which is the number of significant digits you would like to calculate PI to.

PI will constantly display the current step number (2n+1) as well as the number of digits left to calculate. It takes about 10 seconds per decimal during the calculation (depending on the amount of free memory, the number of decimals desired, and other things).

Here are a few decimals of PI:

```
3.1415926535897932384626433832795028841971693993751058209749445923078164
062862089986280348253421170679821480865132823066470938446095505822317253
594081284811174502841027019385211055596446229489549303819644288109756659
334461284756482337867831652712019091456485669234603486104543266482133936
072602491412737245870066063155881748815209209628292540917153643678925903
600113305305488204665213841469519415116094330572703657595919530921861173
819326117931051185480744623799627495673518857527248912279381830119491298
336733624406566430860213949463952247371907021798609437027705392171762931
767523846748184676694051320005681271452635608277857713427577896091736371
787214684409012249534301465495853710507922796892589235420199561121290219
608640344181598136297747713099605187072113499999983729780499510597317328
160963185950244594553469083026425223082533446850352619311881710100031378
387528865875332083814206171776691473035982534904287554687311595628638823
```

PI (# CF6Dh)

```
«
  → P
  «
    1 P 1 + E DUP 2 DIV OVER 5 DIV
    ROT 8 DIV 1 0 0
    → A B C N T S
    «
      A B ADD C ADD
      DO
        T SWAP
        IF
          S
        THEN
          SUBS
        ELSE
          ADD
        END
        'T' STO 1 S - 'S' STO A 4 DIV
        DUP 'A' STO B 25 DIV DUP 'B' STO
        C 64 DIV DUP 'C' STO
        ADD ADD N 2 ADD DUP 'N' STO
        DUP 1 DISP DIV DUP SIZE 2 DISP
      UNTIL
        DUP "0" ==
      END
      DROP T 4 MULT 2 P SUB "3." SWAP +
    »
  »
»
```

VAL

This program evaluates a polynomial at any point. The first argument is the polynomial in vector form; the second is the point (real, complex, algebraic or name). To evaluate x^2+2x+1 at $x=2$, type `[ 1 2 1 ] 2 VAL`

```
VAL (# 2681h)
  «
    → V X
    «
      V SIZE LIST→ DROP → A
      «
        0 1 A
        FOR Y
          V Y GET X A Y - ^ * +
        NEXT
      »
    »
  »
```

DER

This program takes the derivative of a polynomial in vector form. For example, to take the derivative of $3x^2+2x+1$, type `[ 3 2 1 ] DER`

```
DER (# 9063h)
  «
    ARRY→ LIST→ - → A
    « DROP
      IF
        A 0 ==
      THEN
        [ 0 ]
      ELSE
        1 A
        FOR X
          X * A ROLLD
        NEXT
        A 1 →LIST →ARRY
      END
    »
  »
```

A→V and V→A

A→V will convert a polynomial in algebraic form to vector form, and V→A will convert a polynomial in vector form to algebraic form.

Example: '3*X^2+2*X+1' A→V returns [3 2 1].

Note that the program V→A uses the program VAL listed previously.

```
A→V (# 60Dh)
  «
    { } 0 'I' STO
    DO
      0 'X' STO OVER EVAL I FACT / 1 →LIST SWAP +
      SWAP 'X' DUP PURGE ∂ 1 'I' STO+ SWAP
    UNTIL
      OVER 0 SAME
    END
   SWAP DROP 'I' PURGE LIST→ 1 →LIST →ARRY
  »
```

```
V→A (# 4E46h)
  « 'X' VAL »
```

DIVP

This program will calculate the Euclidean division of two polynomials in vector form. For example, to divide the polynomial x^2+2x+1 by the polynomial $x+1$, type: `[ 1 2 1 ] [ 1 1 ] DIVP`. The program will return the quotient in level 2, and the remainder in level 1.

```
DIVP (# 28E3h)
  «
     DUP2 → A B
    «
      B 1 GET A SIZE 1 GET B SIZE 1 GET DUP2 -
      → c n p q
      «
        IF
          p 1 ==
        THEN
          DROP2 A c / [ 0 ]
        ELSE
          0 q
          FOR x
            OVER 1 GET c / DUP 4 ROLLD * n x -
            1 →LIST RDM
            - ARRY→ 1 GET 1 - →ARRY SWAP DROP B
          NEXT
          DROP q 2 + ROLLD q 1 + →ARRY SWAP
        END
      »
    »
  »
```

PCAR

PCAR will calculate the characteristic polynomial of any square matrix. The result is a polynomial in vector form. This vector can then be used with the program LAGU in order to find the roots of the polynomial, which makes it easy to calculate all the correct values of the matrix.

Example: `3 IDN PCAR` returns `[ 1 -3 3 -1 ]` (i.e. x^3-3x^2+3x-1)

PCAR (# DB94h)

```
«
   DUP IDN DUP SIZE LIST→ DROP2 → M I N
   «
      0 N
      FOR X
         M I X * - DET
      NEXT
      N 1 + 1 →LIST →ARRY N 1 + IDN 0 N
      FOR X
         0 N
         FOR Y
            X 1 + N Y - 1 + 2 →LIST X Y ^ PUT
         NEXT
      NEXT /
   »
»
```

LAGU

This program will find all the real and complex roots of any polynomial (which has real or complex coefficients). To use it, place the polynomial on the stack (in vector form) in order of decreasing coefficients of x^i: [a_n ... a_0], the coefficient a_i being the coefficient in front of the term x^i, and execute LAGU. The program will display the different steps of the calculation, and return a list of roots of the polynomial.

LAGU uses Laguerre's algorithm to make the calculation: Z_0 is fixed (an approximation of the root. We can use 0 or the value of the previous root, which saves a lot of time when calculating multiple roots), and calculate $Z_{k+1} = Z_k + S_k$, where S_k is the Laguerre step equal to:

$$S_k = \frac{-nP(Z_k)}{P'(Z_k)+E\sqrt{((n-1)P'(Z_k))^2 - n(n-1)P(Z_k)P''(Z_k)}}$$

In this formula, n is the degree of the polynomial, and P is the polynomial; P' is its first derivative, and P'' is its second derivative. E can be either +1 or -1 to make the denominator as large as possible, in order that the Laguerre step be as small as possible.

Caution: If the polynomial has roots with large multiplicity, the process will oscillate without ever converging. The approximations are best for a polynomial of degree less than 7, and with a maximum multiplicity of 4. LAGU uses the programs VAL, DER, and DIVP previously listed.

Example: To find the roots of x^6-14.x^4+49.x^2-36 , just type:

```
[ 1 0 -14 0 49 0 -36 ] LAGU
```

A few moments later, we get the list of the six roots of the polynomial:

```
{ 1 2 3 -1 -2 -3 }
```

LAGU (# BABFh)

```
«
   IF
      DUP SIZE { 1 } ==
   THEN
      DROP { }
   ELSE
      CLLCD { } 'SOL' STO 0 'Z' STO
      DO
         DUP DUP2 'P' STO 1 DISP 'Z' VAL SWAP DER
         DUP 'Z' VAL SWAP DER 'Z' VAL P SIZE LIST→
         - DUP 1 - DUP SQ 3 PICK 3 PICK * NEG
         → P0 P1 P2 N M A B
         «
            "Root No " N →STR + 2 DISP Z
            WHILE
               DUP 'Z' STO 3 DISP P0 EVAL DUP ABS
               .0000000001 >
            REPEAT
               P1 EVAL P2 EVAL
               → R S T
               «
                  S A S SQ * B R T * * + √ DUP2
                  DUP2 + ABS 3 ROLLD - ABS ≥ 2 *
                  1 - * + DUP
                  IF
                     ABS 0 ==
                  THEN
                     50 .1 BEEP DROP RAND 40 * 20 -
                     RAND 40 * 20 - R→C "/0 New Z0"
                     2 DISP
                  ELSE
                     N NEG R * SWAP / Z +
                  END
               »
            END
            DROP
         »
         SOL Z + 'SOL' STO P 1 Z NEG { 2 }
         →ARRY DIVP DROP
      UNTIL
         DUP SIZE LIST→ ≤
      END
      DROP { Z P } PURGE SOL
   END
»
```

PMAT

This program will calculate the image of any square matrix by a polynomial. It takes the matrix and the polynomial (in vector form) as arguments.

Example: To calculate the image of the identity matrix of order 3 by the polynomial $3x^2+2x+1$, type `3 IDN [ 3 2 1 ] PMAT`

PMAT (# 844Ch)

```
«
   SWAP OVER SIZE 1 GET → V X L
   «
      X 0 CON X IDN L 1
      FOR Y
         DUP V Y 1 →LIST GET * ROT + SWAP X * -1
      STEP DROP
   »
»
```

mSOLVER

mSOLVER will solve a system of non-linear equations containing many unknowns, by using the Newton-Raphson algorithm. To use mSOLVER, you place the various equations to be solved into a list, and store it in 'MEQ'. For example, the following system of equations will find the intersection of two circles. You store it as a list into 'MEQ':

```
{ 'SQ(X)+SQ(Y)=1' 'SQ(X-1)+SQ(Y)=1' } 'MEQ' STO
```

Next, you place the names of the unknowns in a list and store it in 'MVAR'. (In this example: `{ X Y } 'MVAR' STO`) At this point, you may also store approximations in the unknown variables. This step is optional, but it will speed up the solution. For example, you can put 1 into 'X' and 'Y'.

Then place the desired precision on the stack (for example, 0.00001), and execute mSOLVER. During the search, it will display the margin of error of the current calculation. Note that mSOLVER will automatically handle any errors (division by zero, etc.). At the end of the search, it will place different approximations on the stack, "tagged" by the name of the corresponding variable. In our example, we would obtain:

```
4:
3:
2:X: .5
1:Y: .866025403782
```

mSOLVER has two particularities:

- It allows you to find complex roots. To make such a search, simply use complex numbers as an initial approximation.
- It contains many IFERR...END loops, so it is difficult to interrupt the program by pressing [ON]. To stop it, press [ON] twice rapidly.

mSOLVER was written by Christophe Dupont de Dinechin.

JACOB (# E865h)

```
«
  → E V
  «
  'tmp.jacob' CRDIR
  tmp.jacob CLVAR
  { } 1 E SIZE
  FOR e
    1 V SIZE
    FOR v
      E e GET V v GET ∂ +
    NEXT
  NEXT
  UPDIR 'tmp.jacob' PURGE
  »
»
```

μSOLVER (# CC3Dh)

```
«
  CLLCD MEQ MVAR → P E V
  «
    E V JACOB E SIZE V SIZE
    → J SE SV
    «
      DO
        1 SV
        FOR v
          V v GET
          IFERR
            RCL
          THEN
            DROP "Variable Error" 1 DISP 0
          END
          DUP V v GET STO
        NEXT
        SV 1 →LIST →ARRY
        1 SE
        FOR e
          E e GET
          IFERR
            →NUM
          THEN
            "Function Error: ■" ERRM + 1 DISP 0
          END
```

```
        NEXT
        SE 1 →LIST →ARRY
        1 SE
        FOR e
          1 SV
          FOR v
            J e 1 - SV * v + GET
            IFERR
              →NUM
            THEN
              "Jacobian Error: ■" ERRM + 1
              DISP 0
            END
          NEXT
        NEXT
        SE SV 2 →LIST →ARRY
        → X F J
        «
          F X
          IFERR
            F J /
          THEN
            "Singular system: ■" ERRM + 1 DISP
            DROP RAND *
          END
          -
        »
        OBJ→ DROP
        SV 1
        FOR v
          V v GET STO -1
        STEP
      UNTIL
        ABS "Current error: ■" OVER + 1 DISP P ≤
      END
      1 MVAR SIZE
      FOR x
        MVAR x GET DUP RCL SWAP →TAG
      NEXT
    »
  »
»
```

MAZE

In the game MAZE you are lost in the middle of a maze, and you must try to find the exit as quickly as possible.

To play this game, you must begin by entering all the programs that follow . Then, enter the CST menu (by pressing (CST)—found to the left of the (VAR) button). This will activate the 6 menu keys. They each have the following functions:

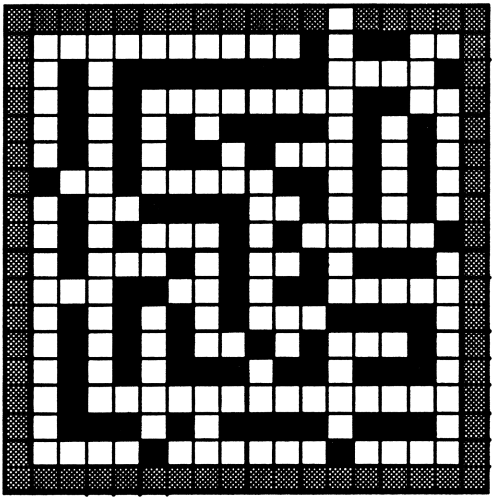
- INIT starts the game. First a maze will be chosen, then the player is placed inside, and the view is displayed on the screen. The x represents your current position.
- VIEW will redraw the current view.
- The four arrows are for moving around in the maze.

In the following listing, only one maze is given. It is possible to add as many others as you wish. The different mazes are contained in a list 'MAZES'. This is a list of lists (one list per maze) which consist of the following:

- A complex number which is the coordinates of the exit.
- A list of 4 binary integers representing the map of the maze.

Coding the map is done in the following way. Each maze is a 16 by 16 grid. Each of the grid boxes can be either a hallway (0) or a wall (1). The map is converted to 4 binary integers. (4 times 64 bits), each one representing a fourth of the maze.

An example is given on the following page.



This diagram is the map of the maze. Each white box represents a section of hallway, each black box a section of wall.

The gray boxes represent "virtual walls"—areas outside the maze. Only one of these boxes is white—the exit—located at coordinates (11,16).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
15	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	15
14	0	1	0	1	1	1	1	1	1	1	1	0	0	0	0	1	14
13	0	1	0	1	0	0	0	0	0	0	0	0	1	1	0	0	13
12	0	1	0	1	0	1	0	1	1	1	1	0	1	0	1	0	12
11	0	1	0	1	0	1	1	0	1	0	0	0	1	0	1	0	11
10	1	0	0	1	0	0	0	0	0	1	1	0	1	0	1	0	10
9	0	1	0	0	1	1	1	1	0	0	1	0	1	0	1	0	9
8	0	1	0	1	0	0	0	1	0	1	0	0	0	0	0	1	8
7	0	1	0	0	0	1	0	1	0	0	1	0	1	1	1	0	7
6	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	0	6
5	0	1	0	1	0	1	0	1	0	1	1	1	1	0	0	0	5
4	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	1	4
3	0	1	0	1	0	1	1	1	0	1	1	1	0	1	1	0	3
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
1	0	1	1	1	0	1	0	1	1	1	1	0	1	1	1	0	1
0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

This table shows the codes for the map. Each section of wall is coded by a 1, each section of hallway by a 0.

The entire maze table can be coded in quadrants, by 4 binary integers, in the following order:

2	4
1	3

Those binary integers would be (ignore the line breaks):

```
1.  # 10100010100110001010101000101010
                    11101010000000101010111000010000b

2.  # 00000000111110100000101010101010
                    01101010000010011111001010001010b

3.  # 01110100000001100111100001000101
                    01110110000000000111011100001000b

4.  # 00110100100001110011000001010111
                    01010001010101100101010010000010b
```

Converting these to hexadecimal, in order, gives the following list:

```
{ # A298AA2AEA02AE10h # FA0AAA6A09F28Ah
  # 7406784576007708h # 3487305751565482h }
```

Here is the listing of programs to enter:

AL (# 5B7Ah)
```
« 0 RDZ 16 RAND * IP »
```

BL1 (# 4998h)
```
"■"
```
(This is not a NEWLINE character but rather ASCII 127, obtained via 127 CHR)

BL2 (# 3D27h)
```
" "
```
(a single space)

TS (# 3E54h)
```
« R→C SO ≠ »
```

TV (# 5A0Dh)
```
« TVP SWAP TVP + »
```

TVP (# 115Fh)
```
« DUP 0 < SWAP 15 > + »
```

ETAT (# 85Ah)
```
«
   DUP2
   IF
      TV
   THEN
      TS
   ELSE
      DUP2 8 / IP SWAP 8 / IP 2 * + 1 + LAB SWAP
      GET 3 ROLLD 8 MOD SWAP 8 MOD SWAP 16 SWAP
      ^ DUP # 1h * * SWAP 2 SWAP ^ * AND # 0h >
   END
»
```

I4 (# AC1h)
```
« X 1 - Y »
```

```
I3 (# 8E47h)
  « X 1 + Y »


I2 (# D48h)
  « X Y 1 - »


I1 (# E9E3h)
  « X Y 1 + »


TEST (# B24Ah)
  «
     1 'COUP' STO+ DUP2
     IF
        TS
     THEN
        DUP2
        IF
           ETAT
        THEN
           "WALL" 1 DISP DROP2
        ELSE
           'Y' STO 'X' STO VIEW
        END
     ELSE
        "BRAVO" 1 DISP DROP2 1400 .1 BEEP
     END
     3 FREEZE
  »


CH (# C52Dh)
  « ETAT 95 * 32 + CHR »


MAZES (# 38FBh)
  {
     {
        (11,16)
        { # A298AA2AEA02AE10h # FA0AAA6A09F28Ah
           # 7406784576007708h # 3487305751565482h }
     }
  }
```

```
CST (# 1FB1h)
  { INIT VIEW
    { "←" GA } { "↑" AV } { "↓" AR } { "→" DR }
  }
```

```
AR (# D13Dh)
  « I2 TEST »
```

```
AV (# A255h)
  « I1 TEST »
```

```
DR (# 7EAh)
  « I3 TEST »
```

```
GA (# 37EDh)
  « I4 TEST »
```

```
VIEW (# 9A77h)
  «
    "         " → S            (that's 9 spaces)
    «
      CLLCD BL1 I1 CH BL1 + + I4 CH BL2 I3 CH
      + + BL1 I2 CH BL1 + + S SWAP + 5 DISP S
      SWAP + 4 DISP S SWAP + 3 DISP "MOVE No "
      COUP + 1 DISP 3 FREEZE
    »
  »
```

```
INIT (# 5E75h)
  «
    MAZES DUP SIZE RAND * 1 + IP GET LIST→ DROP
    'LAB' STO 'SO' STO 1 'COUP' STO 0 0
    DO
      DROP2 AL AL DUP2 ETAT NOT
    UNTIL
    END
    'Y' STO 'X' STO VIEW
  »
```

MASTER

MASTER is the well known game of Mastermind. The object of the game is to try to guess a combination of digits from 0 to 9.

The length of the solution combination can be any size. To set this size, (required to play the first time), simply enter the desired number and execute STOL. Then initialize the game by executing INIT.

To play, you enter a combination of numbers (your guess) in string form, and then execute MASTER. The program will display the number of digits in the right position (Correct) and the number of digits that are in the code, but not in the right position (Found). For example, if the solution is 8548, entering "8834" would return the following:

```
Guess No x

8834

Correct=   1
Found=     2
```

The first 8 is in the right position; the second 8 and the 4 are part of the solution, but are not in the right positions.

To play, enter the programs that follow.

```
STOL (# CF28h)
  «
     DUP
     IF
        TYPE 0 ==
     THEN
        'L' STO INIT
     ELSE
        514 DOERR
     END
  »
```

INIT (# 49F5h)

```
«
   0 'CO' STO 1 L
   START
      RAND 10 * IP
   NEXT
   L →LIST 'SOL' STO
»
```

MASTER (# 28D7h)

```
«
   DUP
   IF
      TYPE 2 == DUP
   THEN
      DROP DUP SIZE L ==
   END
   IF
   THEN
      CLLCD DUP 3 DISP STL PROG 7 FREEZE
   ELSE
      514 DOERR
   END
»
```

STL (# 4DBCh)

```
«
   → S
   «
      { } 1 S SIZE
      FOR X
         S X X SUB STR→ +
      NEXT
   »
»
```

```
PROG(# 743Fh)
 «
   0 0
   → PR CP CT
   «
     1 'CO' STO+ "Guess No " CO + 1 DISP
     SOL PR 1 L
     FOR X
       DUP X GET 3 PICK X GET
       IF
         ==
       THEN
         X -2 PUT SWAP X -1 PUT SWAP 1 CP +
         'CP' STO
       END
     NEXT
     'PR' STO "Correct=     " CP + 5 DISP 1 L
     FOR X
       DUP X GET DUP
       IF
         -1 >
       THEN
         1 L
         FOR Y
           PR Y DUP2 GET 4 PICK
           IF
             ==
           THEN
             -2 PUT 'PR' STO 1 CT + 'CT' STO
             4 'Y' STO
           ELSE
             DROP2
           END
         NEXT
       END
       DROP
     NEXT
     DROP "Found=         " CT + 6 DISP
   »
 »
```

ANAG

This program takes a string of characters and displays all possible anagrams. For example, "ABC" ANAG will display these character strings: "ABC" "ACB" "BAC" "BCA" "CAB" "CBA". Here are the programs:

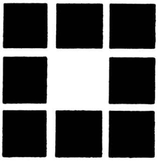
```
PRANAG(# A68Dh)
  «
    IF
      B 0 >
    THEN
      -1 'B' STO+ PRDEPTH DUP B -
      FOR X
        X ROLL PRANAG X ROLLD -1
      STEP
      1 'B' STO+
    ELSE
      PRDEPTH DUPN PRDEPTH 2 / 1 - 1
      START
        + -1
      STEP
      4 DISP
    END
  »
```

```
PRDEPTH(# EAFFh)
  « DEPTH C - »
```

```
ANAG (# 1F82h)
  «
    → A
    «
      CLLCD A SIZE 'B' STO DEPTH 'C' STO 1 B
      FOR X
        A X DUP SUB
      NEXT
      PRANAG PRDEPTH DROPN { B C } PURGE
    »
  »
```

SQUARE

The goal of this game is to arrive at a display of the "magic square," which is the following figure:



To accomplish this, the player may press dif ferent boxes (by using the keys [1] to [9]). Pressing one of these will inverse the box as well as some of its neighbors.

To play, enter the following programs, and execute 'SQUARE'.

```
KEYS (# 2CE6h)
  { 82 83 84 72 73 74 62 63 64 }

MESS (# 8D19h)
  "WORKING..."

T (# 6459h)
  {
    { 1 2 4 5 } { 1 2 3 } { 2 3 5 6 }
    { 1 4 7 } { 2 4 5 6 8 } { 3 6 9 }
    { 4 5 7 8 } { 7 8 9 } { 5 6 8 9 }
  }

M (# EE2h)
  { "      789     →" "         456     →"
    "      123     →" }
```

```
CALC (# E30Ah)
  «
    "Press a key..." 1 DISP T 1
    DO
      DROP KEYS
      DO
      UNTIL
        KEY
      END
    UNTIL
      POS DUP
    END
      1000 .05 BEEP MESS 1 DISP GET DUP 1 DUP
      ROT SIZE
    START
      GETI 1 - DUP 3 MOD 1 +
      WHILE
        DUP 3 >
      REPEAT
        3 -
      END
      SWAP 3 / IP 1 + SWAP 2 →LIST CAR SWAP
      DUP2 GET NOT PUT 'CAR' STO
    NEXT
    DROP2
  »


SOL (# C888h)
  [[ 1 1 1 ]
   [ 1 0 1 ]
   [ 1 1 1 ]]


CAR (# C888h)
  [[ 1 1 1 ]
   [ 1 0 1 ]
   [ 1 1 1 ]]
```

```
VISU(# E530h)
  «
    DO
      CAR { 1 1 } 1 3
      FOR X
      "" 1 3
        START
          3 ROLLD GETI 95 * 32 + CHR 4 ROLL
          SWAP +
        NEXT
        M X GET SWAP + 142 CHR + 3 ROLLD
      NEXT
      DROP2 2 4
      FOR X
        X 1 + DISP
      NEXT
      CALC
    UNTIL
      CAR SOL ==
    END
      "           Bravo..." 1 DISP 1 3
    START
      1000 .2 BEEP
    NEXT
  »
```

```
SQUARE(# 2DC2h)
  «
    CLLCD MESS 1 DISP 0 RDZ CAR
    DO
      { 1 1 } 1 9
      START
        RAND .5 > PUTI
      NEXT
      DROP DUP
    UNTIL
      SOL ≠
    END
    'CAR' STO VISU
  »
```

PR40

This program will print character strings with 40 characters per line instead of 24 on the HP 82240A or HP 82240B infrared printer. The string may contain carriage returns (■), and any line which contains more that 40 characters is split (just like the function PR1).

The program is simple. It takes the string and splits it, first at each carriage return, then it cuts the portions that are longer than 40 characters. Each of the sections thus obtained are transformed into graphics objects in the smallest font (using 1 →GROB) and then printed using the function PR1. Because of this, any small letters are changed to capitals.

This program is particularly useful for printing listings obtained from the disassembler.

```
PR40 (# 7B55h)
  « "■" + → S
    «
      WHILE
        S SIZE
      REPEAT
        S DUP "■" POS DUP2 1 + OVER SIZE
        SUB 'S' STO 1 SWAP 1 - SUB
        → T
        «
          WHILE
            T SIZE
          REPEAT
            T 1 40 SUB 1 →GROB PR1 DROP T 41
            OVER SIZE SUB 'T' STO
          END
        »
      END
    »
  »
```

DSP and INITSCR

These two programs, DSP and INITSCR, let you use the HP 48 screen in 33-column mode. The display is shown line-by-line to allow you to see each line while it is being displayed.

The two programs perform the following functions:

- INITSCR erases the screen and initializes the screen memory used for the line-by-line display.
- DSP displays the message line-by-line scrolling up any text already displayed.

The function →GROB is used to obtain the small font characters. A graphics object is created for each line of the display, and then each line is saved, in list form, in a variable called SCREEN. The lines are added using the OR function on a blank GROB, and then the result is displayed using the →LCD function.

This program can be used with the program DISASM (the disassembler) to view the listing as it is disassembled. DSP can replace the RPL function DISP. To do this, replace 1 DISP in the program STOS with DSP and add INITSCR to the beginning of the program DISASM.

```
INITSCR(# 424h)
  « { } 'SCREEN' STO CLLCD »
```

```
DSP (# 70A4h)
  «
    IF
      "■" OVER DUP SIZE DUP SUB OVER ≠
    THEN
      +
    ELSE
      DROP
    END
    → TXT
    «
      WHILE
        TXT 1 OVER "■" POS DUP
      REPEAT
        3 DUPN SWAP + OVER SIZE SUB 'TXT' STO
        1 - SUB 1 →GROB SCREEN + 1 9 SUB
        'SCREEN' STO # 83h # 40h BLANK 1 SCREEN
        SIZE DUP # 6h *
         → O
        «
          FOR X
            # 0h O # 6h X * - 2 →LIST SCREEN
            X GET GOR
          NEXT
        »
        →LCD
      END
      3 DROPN
    »
  »
```

MUSICML

MUSICML will play tunes without interruptions between notes. MUSICML is a machine language program that has not yet been assembled. The RPL program listed below will take a list of notes (frequency, duration) and create a machine language program that will play them. It uses the two programs GASS and A→STR, previously listed.

Example: { 1400 .1 2800 .1 1400 .1 } MUSICML EVAL

Note: The 'Code' object (which is on the stack before executing EVAL) can be stored in a variable to be used later.

The following is the RPL listing of MUSICML; the disassembled source listing of the machine language portion is given on the next page.

```
MUSICML(# EC8h)
  «
    → L
    «
      "CCD20" # 4Fh L SIZE 2 + 5 * + A→STR +
      "8FB97608E" + L SIZE 2 + 5 * A→STR 1 4
      SUB + 1 L SIZE
      FOR X
        L X GET A→STR + L X 1 + GET 1000 *
        A→STR + 2
      STEP
     "0000000000071351 47D71741431741370 6D68AAD08F6A7"
       "1069DF078F2D760142164808C" + + GASS
    »
  »
```

```
      CCD20    CON(5)   PROL_CODE        Code object
start *****    CON(5)   (end)-(start)    Code length
      8FB9760  GOSBVL   SAVE_REG         Backup regs.
      8E****   GOSUBL   l1
```

LIST OF NOTES—Frequency / Duration (in milliseconds)

```
      CON(5)   #00000                    End of notes
      CON(5)   #00000
l1    07       C=RSTK
      135      D1=C
      147      C=DAT1   A                Read frequency
      D7       D=C      A
      174      D1=D1+   5
      143      A=DAT1   A                Read duration
      174      D1=D1+   5
      137      CD1ex
      06       RSTK=C
      D6       C=A      A
      8AA      ?C=0     A                Done?
      D0       GOYES    l2
      8F6A710  GOSBVL   BEEP_LM          Beep
      69DF     GOTO     l1               Loop
l2    07       C=RSTK
      8F2D760  GOSBVL   LOAD_REG         Restore regs.
      142      A=DAT0   A                Return to RPL
      164      D0=D0+   5
      808C     PC=(A)
end
```

MODUL

This machine language program will quickly alternate between two sound frequencies. The arguments are a starting frequency (START), an ending frequency (END), a frequency increment (INCREMENT), and the duration of each note (DUR). These settings are used by the RPL program MODUL, which automatically creates a machine language program that will make the sound. This program uses GASS and A→STR, listed previously.

Example: `1400 2800 50 .01 MODUL EVAL`

Note: The 'Code' object (which is on the stack before executing EVAL) can be stored in a variable to be used later.

Here is the commented assembly source listing for the assembly routine created by MODUL. The asterisks (*) represent code that depends on one of the 4 arguments.

```
      CCD20     CON(5)    PROL_CODE        Code object
start 15000     CON(5)    (end)-(start)    Code length
      8FB9760   GOSBVL    SAVE_REG         Backup regs.
      34*****   LCHEX     START            Start frequency
      D7        D=C       A
l1    DB        C=D       A
      06        RSTK=C
      34*****   LCHEX     DUR              (In milliseconds)
      8F6A710   GOSBVL    BEEP_LM          Beep
      07        C=RSTK
      D7        D=C       A
      34*****   LCHEX     INCREMENT        Increment
      C3        D=D+C     A
      34*****   LCHEX     END              Ending frequency
      ***       ?D<=C     A                Or: ?D>=C  A
      7D        GOYES     l1               Loop
      8F2D760   GOSBVL    LOAD_REG         Restore regs.
      142       A=DAT0    A                Return to RPL
      164       D0=D0+    5
      808C      PC=(A)
end
```

```
MODUL (# 1E1Fh)
  «
    → D F I P
    «
      IF
        P 1000 * DUP 'P' STO # 0h + # 0h ==
      THEN
        "ZERO DURATION..." DOERR
      END
      IF
        I # 0h + # 0h ==
      THEN
        "ZERO INCREMENT..." DOERR
      END
      "CCD20150008FB976034" D A→STR + "D7DB0634" +
      P A→STR + "8F6A71007D734" + I A→STR +
      IF
        D F <
      THEN
        "C3"
      ELSE
        "E3"
      END
      + "34" + F A→STR +
      IF
        D F <
      THEN
        "8BF"
      ELSE
        "8BB"
      END
      + "7D8F2D760142164808C" + GASS
    »
  »
```

RABIP

This little program will generate random sounds in the frequency range of 0 to 4400 Hz, for a duration of 0 to 0.1 seconds each. It stops when any key is pressed. This could be used as an original way of letting the user know that some long program has finished its calculations.

```
RABIP (# A75Bh)
  «
    DO
      4400 RAND * .1 RAND * BEEP
    UNTIL
      KEY
    END
    DROP
  »
```

JINGLE

This program plays a little music. The notes for the tune are contained in the list SOUNDS (an example is given here). Note that the SOUNDS list is given in reverse. The last frequency-duration pair is the first note played.

```
JINGLE (# 83E1h)
  «
    SOUNDS LIST→ 1 SWAP 2 / MEM DROP
    START
      BEEP
    NEXT
  »
```

```
SOUNDS (# 9A73h)
  { 390 .75 440 .15 275 .15 350 .075 350 .15 390
    .075 690 .15 565 .15 390 .15 465 .15 565 .15
    590 .075 390 .075 390 .15 565 .3 390 .3 350
    .15 390 .15 515 .075 390 .075 390 .15 465 .3
    390 .3
  }
```

RENAME

This program allows you to rename an object. It takes the old name and the new name as arguments. The object is renamed without changing its position in the directory order.

RENAME (# 1A24h)

```
«
  OVER RCL SWAP STO VARS DUP2 SWAP POS 2 SWAP
  SUB ORDER PURGE
»
```

AUTOST

AUTOST is an example autostart program. You may add to this program to improve it as you wish. As is, this program will be assigned to the [OFF] key automatically (i.e. it will make the assignment and put the calculator into USER mode).

AUTOST (# BCE5h)

```
«
  «
    CLLCD OFF 1400 .07 BEEP "HP48 : READY"
    1 DISP
    1000 .01 BEEP .5 WAIT
  »
  91.3 ASN -62 SF
»
```

CAL

CAL will display a one month calendar. As arguments, it takes a list of two real numbers that specify the month to display: The number of the month (between 1 and 12) and the year (between 1583 and 9999).

Or, a quicker method:

- If the list contains only one element, this is considered to be the month number, and the year will be the current year according to the calculator clock.
- If the list is empty, then the current month is displayed.

Note that the calendar is "European" style; Monday is the first day of the week.

```
CAL (# 2E31h)
 «
   CLLCD # 4E2CFh SYSEVAL RCLF
   → F
   «
     -42 SF { } + DATE FP 100 * SWAP OVER IP +
     SWAP FP 10000 * + DUP DUP SIZE 2 MOD 2 +
     GET SWAP 1 GET
     → Y M
     «
       1.0119 1 M 100 / + Y 1000000 / + DDAYS
       7 MOD
       → S
       «
         { "JANUARY" "FEBRUARY" "MARCH" "APRIL"
           "MAY" "JUNE" "JULY" "AUGUST"
           "SEPTEMBER" "OCTOBER" "NOVEMBER"
           "DECEMBER"
         }
         M GET " " + Y + "                 "
         1 22 4 PICK SIZE - 2 / SUB SWAP +
         1 DISP " MO TU WE TH FR SA SU" 2 DISP
         { 31 28 31 30 31 30 31 31 30 31 30
           31 }
```

```
            M GET M 2 == Y 4
            MOD 0 == Y 100 MOD 0 == - Y 1000 MOD
            0 == + AND +
            → N
            «
              0 5
              FOR L
                "" 1 7
                FOR C
                  L 7 * C + S - "    " SWAP
                  IF
                    DUP 0 > OVER N ≤ AND
                  THEN
                    +
                  ELSE
                    DROP
                  END
                  DUP SIZE DUP 2 - SWAP SUB +
                NEXT
                L 16 * # 1247Bh + SYSEVAL
              NEXT
              7 FREEZE
            »
          »
        »
      »
    »
```

CIRCLE

CIRCLE is a rapid circle drawing routine written by Christophe Nguyen. It uses the Bresenham algorithm and takes two arguments: A real number, the diameter of the circle (if the diameter is negative, a white circle with a diameter of that absolute value is drawn); and a complex number , the coordinates of the center of the circle. These two arguments are left on the stack. If they are no longer useful, you should drop them (with DROP2).

This program is self-modifying; it should not be used as a backup (saved in a port). Three demonstration programs (TEST1, TEST2, and TEST3) show how fast it is. Its long disassembled source listing is omitted here.

TEST1 (# D683h)

```
«
   ERASE { # 0h # 0h } PVIEW 1 1000
   START
      RAND 20 * RAND 131 * 65 - RAND 64 * 32 -
      R→C CIRCLE DROP2
   NEXT
»
```

TEST2 (# 58EEh)

```
«
   ERASE { # 0h # 0h } PVIEW 10 (0,0) 1 20
   START
      CIRCLE DUP2 RAND 10 * 5 - RAND 10 * 5 -
      R→C + CIRCLE
   NEXT
   1 1000
   START
      DUP2 RAND 10 * 5 - RAND 10 * 5 - R→C +
      CIRCLE DEPTH ROLL -1 * DEPTH ROLL CIRCLE
      DROP2
   NEXT
»
```

TEST3 (# 35EFh)

```
«
   INIT DEG
   DO
```

```
        -180 180
        FOR T
           5 T * COS 60 * 7 T * SIN 30 * R→C 3
           OVER CIRCLE DROP2 DEPTH ROLL -3 SWAP
           CIRCLE DROP2 2
        STEP
     UNTIL
        0
     END
  »
```

```
INIT(# 50F1h)
  «
     ERASE 1 20
     START
        (100,100)
     NEXT
     { # 0h # 0h } PVIEW
  »
```

```
CIRCLE(# 9965h)
  D9D20 2ABF1 3FBF1 CCD20 99300 8FB97 60201 37135
  06147 13517 41371 35067 42110 B0713 517F7 51110
  C0706 13517 41471 35174 13713 575F0 10807 13517
  41471 35174 17E20 15719 1A511 10F81 00808 21716
  A0080 82190 70000 7D534 4D200 C9137 1491B 56507
  14213 216EA F0142 8A867 70000 7D534 A6200 C9137
  14181 C1CD1 41AF0 142CC 81C81 CE4E4 81C1C F1C01
  CF141 184AF 01428 A8721 CD141 81C1C D1411 6917F
  17F17 41321 417BB 0208F 2D760 14216 4808C 13713
  51341 6ED21 53332 0059B E0332 4009B 2A032 003AB
  AAB61 7D133 EA131 80D0D 21571 6900D 22001 23D1D
  71F81 C0013 37F20 1FC90 00133 71201 F6000 01337
  310DB E9152 19184 0FA20 01A8B 90AB0 C8A0E 65FF0
  D01D0 10111 0102C 43430 000E2 10811 111AE A2430
  F9026 06160 7B601 10243 0F906 D1119 C6C6C A2034
  60000 CA100 69201 1AD51 19E9C 6C6CA 2034A 0000C
  A1001 1ACE1 0A119 E6109 6F8F1 1111A 8A600 72000
  18407 54011 9D511 A109D 910A7 130F9 D910A 76201
  19FA1 0AF9D 91097 310F9 D9109 87000 8506A BF201
  1A808 24020 00EA1 1CEA3 40400 08BE0 0C434 11000
  D7340 3217C F480C 268FF 13711 13414 000CA 11BCA
  34380 008BE 00D6A F0DAA E781C 81C13 7C213 5A64A
  64AEB B62AE A301A 6C490 A1666 FF153 10E1E 15110
  11531 B9C0E 1EB9C 15110 1B213 0
```

BANNER

The program BANNER will allow you to display a scrolling message in giant letters. BANNER was written by Christophe Nguyen.

Notes:

- The accepted characters are the ASCII characters from 31 to 90 (numbers, punctuation, and capital letters).
- BANNER uses a table to draw the characters. Because this table needs to be generated by the program MKT, entering the programs is a little different than usual. To enter BANNER, do the following:
  - Enter the code for BANNER1, as a string on one line with no spaces, and place it on the stack.
  - Enter and execute the program MKT (which will produce a string of 2100 characters).
  - Enter the code for BANNER2, as a string on one line with no spaces, and place it on the stack.
  - Concatenate the three strings (by pressing ⊕ twice).
  - Execute GASS (or RASS) and store the result as 'BANNER'. The resulting program should look like 0 CHR + CLLCD Code DROP.

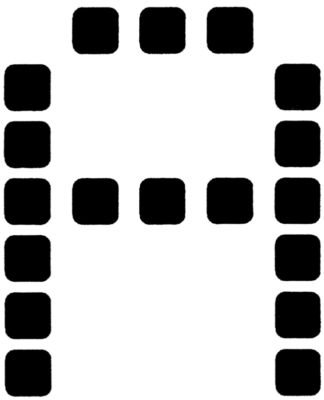
To use BANNER, simply give it a string of characters, and watch the results. Example: "JOURNEY TO THE CENTER OF THE HP48..." BANNER

Here is the commented assembly source listing for BANNER, then the codes for BANNER1, and BANNER2, and the program MKT:

```
      D9D20    CON(5)   PROL_PRGM        Program object
      4B2A2    CON(5)   #2A2B4           Null
      66BC1    CON(5)   #1CB66           CHR
      76BA1    CON(5)   #1AB67           Addition
      858A1    CON(5)   #1A858           CLLCD
      CCD20    CON(5)   PROL_CODE        Code object
start 23A00    CON(5)   (end)-(start)    Code length
      8FB9760  GOSBVL   SAVE_REG         Backup regs.
      1BE0507  D0=(5)   7050E
      142      A=DAT0   A                A=@ screen bitmap
      3412000  LCHEX    #00021
      C2       C=C+A    A
      134      D0=C
      10A      R2=C                      Current position
      137      CD1ex
      135      D1=C
      06       RSTK=C
      AE0      A=0      B
      8082180  LAHEX    #08              Big pixel height
      100      R0=A
      AE0      A=0      B
      8082120  LAHEX    #02              Big pixel width
      101      R1=A                      (2 nibbles,8 bits)
      07       C=RSTK
      135      D1=C
      143      A=DAT1   A                A=@ string
      131      D1=A
      179      D1=D1+   10               D1=@ of first char.
      137      CD1ex
      135      D1=C
      06       RSTK=C
Loop  1BE0507  D0=(5)   7050E
      142      A=DAT0   A
      130      D0=A                      D0=@ screen bitmap
      16F      D0=D0+   16
      16F      D0=D0+   16
      160      D0=D0+   1                ds screen position
      07       C=RSTK
      135      D1=C                      D1=@ char.
      D0       A=0      A
      14B      A=DAT1   B                Read 1 char.
      96C      ?A#0     B                CHR(0)?
      80       GOYES    Cont             Continue
      8CD990   GOLONG   Done             Done
```

```
Cont 34F1000   LCHEX    #0001F       ;
     EE        C=A-C    A            ;Calculate the
     DA        A=C      A            ;offset to
     C6        C=C+C    A            ;find the
     C6        C=C+C    A            ;representation of
     C6        C=C+C    A            ;the char.
     C6        C=C+C    A            ;1 char= 35 data
     C6        C=C+C    A            ;Char between 31
     C2        C=C+A    A            ;and 90, then:
     C2        C=C+A    A            ;offset=(c-31)*35
     C2        C=C+A    A            ;
     DA        A=C      A            ;
     8E4380    GOSUBL   Get_code      Gosub after the data
                                      (to determine address)
```

```
* End of BANNER1 and beginning of the character codes
*
* These codes are 1 nibble per pixel (0 or F) to speed up
* execution. They are coded column by column.
*
* Take, for example, the letter A:
*
*
```



```
*      0 F F F 0
*      F 0 0 0 F
*      F 0 0 0 F
*  ->  F F F F F
*      F 0 0 0 F
*      F 0 0 0 F
*      F 0 0 0 F
*
* The code for A looks like:
*            0FFFFFFF00F000F00F000F00F0000FFFFFF...
*
* End of character codes, and beginning of BANNER2
```

```
Get_code
     07       C=RSTK                C=@ of data
     CA       A=A+C      A          Add offset
     171      D1=D1+     2          @next char
     137      CD1ex
     06       RSTK=C                Save
     131      D1=A
     305      LCHEX      #5         5 columns
     A97      D=C        WP
Next A1F      D=D-1      WP
     560      GONC       St_col     If not done
     6F70     GOTO       Blank      Otherwise --> blank
St_col
     3170     LCHEX      #07        7 lines
     AE5      B=C        B
Wr_col
     A6D      B=B-1      B
     472      GOC        End_col    Done
     1531     A=DAT1     WP         Read pixel
     118      C=R0                  Big pixel height
Repeat_H
     AIE      C=C-1      WP
     431      GOC        End_repH   Done
     1501     DAT0=A     WP         Write
     16F      D0=D0+     16         ; Go to the
     16F      D0=D0+     16         ; next line
     161      D0=D0+     2          ;
     6CEF     GOTO       Repeat_H
End_repH
     170      D1=D1+     1          Next big pixel
     68DF     GOTO       Wr_col
End_col
     119      C=R1                  We have written on
     AE5      B=C        B             right of screen:
     1BE0507  D0=(5)     7050E         Now we must
     146      C=DAT0     A             scroll to the left
     134      D0=C
     118      C=R0                  Recalculate the num-
     AEA      A=C        B             ber of lines to
     A64      A=A+A      B             scroll.
     A64      A=A+A      B
     A64      A=A+A      B
     B6E      C=A-C      B
```

```
Repeat_L
     A6D      B=B-1    B           Extension of
     471      GOC      Next_col      width
     1BE0507  D0=(5)   7050E
     142      A=DAT0   A
     130      D0=A
     7D50     GOSUB    Left        Scroll
     68EF     GOTO     Repeat_L
Next_col
     11A      C=R2
     134      D0=C
     6D7F     GOTO     Next
Blank11A      C=R2                 Adding space between
     134      D0=C                   two characters
     118      C=R0
     AEA      A=C      B
     A64      A=A+A    B
     A64      A=A+A    B
     A64      A=A+A    B
     B6E      C=A-C    B
     AE5      B=C      B
     A90      A=0      WP
Part A6E      C=C-1    B           Write a blank column
     431      GOC      Leftbl
     1501     DAT0=A   WP
     16F      D0=D0+   16
     16F      D0=D0+   16
     161      D0=D0+   2
     6CEF     GOTO     Part
LeftblAE9     C=B      B           Scroll
     1BE0507  D0=(5)   7050E
     142      A=DAT0   A
     130      D0=A
     7600     GOSUB    Left
     8C896F   GOLONG   Loop
Left 06       RSTK=C               Scroll the visible part
     AE4      A=B      B           of the display.
     AE5      B=C      B
     AE6      C=A      B           C= # of lines
     06       RSTK=C               D0=@ screen bitmap
```

```
Loop_lft
        A6D      B=B-1     B
        571      GONC      Nextlft
        07       C=RSTK
        AE5      B=C       B
        34BB000  LCHEX     #000BB
Wait    CE       C=C-1     A           Delay to slow scrolling
        5DF      GONC      Wait
        07       C=RSTK
        01       RTN                   Done.
Nextlft
        2F       P=        15          Scroll one single line
        1521     A=DAT0    WP
        B94      ASR       WP
        1501     DAT0=A    WP
        16E      D0=D0+    15
        1521     A=DAT0    WP
        B94      ASR       WP
        1501     DAT0=A    WP
        16E      D0=D0+    15
        142      A=DAT0    A
        F4       ASR       A
        1503     DAT0=A    X
        20       P=        0
        163      D0=D0+    4           Next line
        68BF     GOTO      Loop_lft    Continue
Done    8F2D760  GOSBVL    LOAD_REG    Restore regs.
        142      A=DAT0    A           Return to RPL
        164      D0=D0+    5
        808C     PC=(A)
end     8DBF1    CON(5)    DROP
        B2130    CON(5)    EPILOGUE
```

Here are the programs that you will need to enter. *The method of entering these is not the same as usual. Please read the notes on page 324.*

```
BANNER1 (# 4C06h)
  D9D20 4B2A2 66BC1 76BA1 858A1 CCD20 23A00 8FB97
  601BE 05071 42341 2000C 21341 0A137 13506 AE080
  82180 100AE 08082 12010 10713 51431 31179 13713
  5061B E0507 14213 016F1 6F160 07135 D014B 96C80
  8CD99 034F1 000EE DAC6C 6C6C6 C6C2C 2C2DA 8E438
  0
```

```
MKT (# DF20h)
  «
    "" { # 0h # 0h } PVIEW 31 90
    FOR A
      PICT { # 0h # 0h } A CHR 2 →GROB REPL 0 4
      FOR X
        0 6
        FOR Y
          IF
            X R→B Y R→B 2 →LIST PIX?
          THEN
            "F"
          ELSE
            "0"
          END
          +
        NEXT
      NEXT
    NEXT
  »
```

```
BANNER2 (# B995h)
  07CA1 71137 06131 305A9 7A1F5 606F7 03170 AE5A6
  D4721 53111 8A1E4 31150 116F1 6F161 6CEF1 7068D
  F119A E51BE 05071 46134 118AE AA64A 64A64 B6EA6
  D4711 BE050 71421 307D5 068EF 11A13 46D7F 11A13
  4118A EAA64 A64A6 4B6EA E5A90 A6E43 11501 16F16
  F1616 CEFAE 91BE0 50714 21307 6008C 896F0 6AE4A
  E5AE6 06A6D 57107 AE534 BB000 CE5DF 07012 F1521
  B9415 0116E 1521B 94150 116E1 42F41 50320 16368
  BF8F2 D7601 42164 808C8 DBF1B 2130
```

Appendices

A. Answers to Exercises

1-1. [←][-] (the left-shifted [0] key)

1-2. [→][RCL] (the right-shifted [STO] key)

2-1. One possible sequence is [5][ENTER][3][ENTER] [1][+][9][ENTER] [5][−] [×][÷]. (With some functions, like [+], [−], and [×], you don't need to press [ENTER] after pressing them).

2-2. For example `SWAP ROT`

2-3. `COS((3*5)-11)/4-1)` which gives `1 (COS(0))`.

3-1. Type [→][HOME] ['][E][X][O][ENTER] [←][MEMORY] [CRDIR] [VAR] [EXO] [1] ['][A] [STO] [2]['][B] [STO] [3]['][C] [STO]

3-2. 6 (PARTS, PROB, HYP, MATR, VECTR, and BASE).

4-1. `« → A B « A B + » »` This can also be used to add two strings.

4-2. It calculates the fraction (A+B)/(A*B) where A and B are two real numbers taken from the stack.

4-3. An example:

```
FIBO(# 5B7Eh)
  «
    → N
    «
      IF
        N 1 ≤
      THEN
        1
      ELSE
        N 1 - FIBO N 2 - FIBO +
      END
    »
  »
```

5-1. 1h, Ah, 19h, FFFFh, BEBEh.

5-2. 291, 16, 256, 2898, 3.

6-1. B73, AFB.

6-2. For **P**: B03 and A8B ; for **WP**: B13 and A9B.

6-3. `A13 D=D+C WP, A73 D=D+C W, A83 D=0 P, A93 D=0 WP.`

6-4. 411.

6-5. 41.

6-6. C411.

6-7. #70080h:0, #70081h:1, #70082h:2.

6-8. **C** field **X** contains 210, **C** field **B** contains 10, and **C** field **XS** contains 2.

6-9. 3 (the nibbles 0, 1, and 2).

7-1. The program codes are as follows:

```
CCD20            CON(5)     #02DCC
45000     begin  CON(5)     (end)-(begin)
6310             GOTO       l1
CC        sub1   A=A-1      A
3454321          LCHEX      #12345
CE        l2     C=C-1      A
5DF              GONC       l2
03               RTNCC
3450000   l1     LCHEX      #00005
DA               A=C  A
7CEF      l3     GOSUB      l2
8AC              ?A#0       A
9F               GOYES      l3
3410000          LCHEX      #00001
DA               A=C        A
7110             GOSUB      l4
8A8              ?A=0       A
40               GOYES      l5
CC               A=A-1      A
142       l5     A=DAT0     A
164              D0=D0+     5
808C             PC=(A)
```

```
8AA     l4   ?C=0      A
00           RTNYES
D2           C=0       A
E4           A=A+1     A
01           RTN
        end
```

The code listing would look like this:

```
CCD20 45000 6310C C3454 321CE 5DF03 34500 00DA7
CEF8A C9F34 10000 DA711 08A84 0CC14 21648 08C8A
A00D2 E401
```

7-2. The listing decodes to:

```
143     A=DAT1    A
133     AD1EX
179     D1=D1+    10
1577    C=DAT1    W
B76     C=C+1     W
1557    DAT1=C    W
131     D1=a
142     A=DAT0    A
164     D0=D0+    5
808C    PC=(A)
```

8-1. The system binary <54321h>.

8-2. 11920 EDCBA

8-3. 11920 B7000

8-4. 33920 100 000000000021 0

8-5. -77345.

8-6. Some precision would be lost by coding it as 55920 51000 543210987654321 0.

8-7. -1E-2 (-0,01).

8-8. 77920 000 000000000001 0 10000 0000000002 0

8-9. (-33, 33).

8-10. D9920 00000 00000000000000 0 00000...
...00000000000000 0

8-11. The long complex (1.23456789012345, -543210987654321)

8-12. FB920 34

8-13. The character 'D' (ASCII code 44h).

8-14. 8E920 67200 11920 30000 30000 50000 80000...

8-15. It contains character strings.

8-16. C2A20 B1000 84 56 C6 C6 F6 02 75 F6 27 C6 46

8-17. "Bravo !"

8-18. E4A20 51000 1BF7935000000000

8-19. #54321h

8-20. 47A20B2130

8-21. { OK }

8-22. 69A20 FF7 12000 00000 10 44 10 C2A207000033
21000 10 14 10 C2A207000043

8-23. 69A20 100 321 03EF7 00000 12000 00000 10 44 10
C2A207000033 21000 10 14 10 C2A207000043

8-24. 8BA20 84E201014 84E201024 76BA1 B2130

8-25. 'A*(B-C)'

8-26. ADA20 33920000000000000210 C2A2070000D6
68B01 B2130

8-27. 5.1_m^3

8-28. `CFA20 20 55E4 84E2030451474`

8-29. `OK:CORRAL`

8-30. `GROB 4 1 F0`

8-31. #6FFh

8-22. `'VIDE'`

8-33. No.

8-34. `'BCKP'`

8-35. #62D6h

8-36. With #2361Eh and #23639h.

8-37. With #1AB67h.

8-38. `CCD20 50000`

8-39. `CCD20 F0000 142 164 808C`. This is the program, which does nothing but pass control to the next object:

```
142          A=DAT0     A
164          D0=D0+     5
808C         PC=(A)
```

8-40. `84E20 50 84 56 C6 C6 F6`.

8-41. An empty name.

8-42. 4

8-43. `'Name'`.

8-44. `29E20 654 321`

8-45. Library #001h, command #002h.

B. Background Information

Manufacturing Information

To determine the version number of your machine, turn the HP48 on, press and hold down the [ON] key. While holding that down, press [D]. Now release [D], then release [ON]. Three lines should show on the screen.

Now press backspace ([←]). The text "705D9:1B8DA178E5A111B6" should appear at the top of the screen. Now press [EVAL]. You should see something similar to this:

```
Version HP48-?
Copyright HP 1989
```

The *?* is your ROM version (A, B, C, D, E, etc.). To return to the normal state, press the buttons [ON]-[C] (just as you pressed [ON]-[D]).

When and where was your HP48 manufactured? The serial number (on the back of the calculator, above the battery compartment) tells you:

- The first two digits show the number of years since 1960.
- The next two digits are the week number of that year.
- Then comes the initial of the country where the machine was manufactured (A for America, B for Brazil, S for Singapore).
- The last 5 digits tell its manufacturing order for that week.

Thus, for example, the HP 48 with serial number 3007A01051 was the 1051st machine made in America during the 7th week of 1990.

Troubleshooting

When your HP48 is locked up (i.e. it doesn't seem to respond to any key presses) try, in this order, these possible solutions:

- [ON] will interrupt the majority of programs in execution without danger of losing memory.
- [ON]-[C] is a system reset, or "warm boot", and will not af fect memory (except the stack is lost).
- [ON]-[A]-[F] will erase the memory. You will be asked the question, `Try To Recover Memory?`. At this point you can either answer YES, or NO. This restoration can fail if there are serious problems with RAM. This restoration can sometimes cause the machine to lock up, so you will need to use the next solution given here.
- On the bottom of the HP48 are 4 rubber feet that are not glued in, so they can be removed and replaced easily. Underneath one of the feet near the top (either the left or the right, depending on the model), you will find a little hole with the letter 'R' next to it (as in RESET). By inserting a thin object, (like a paper clip), you can press a reset button inside. If you only press it for a short while, the User data will be preserved. By pressing it for longer (one or two seconds), the HP48 memory will be completely erased. CAUTION: this button is fragile. Do not use this method unless absolutely necessary.
- As a last resort, you can remove the batteries. There are some capacitors inside the calculator that still give it power even when the batteries are out, so you will need to discharge them. Two solutions are possible: wait a few hours, or insert the batteries backwards for a few seconds (there is no danger, the HP48 is protected with diodes). Then insert the batteries properly and turn it on.
- If none of the methods listed above work, then the best thing to do is to return the calculator to an authorized Hewlett-Packard dealer for repairs.

Binary, Hexadecimal, and Other Barbarities

Here are a few principles that you will need to know well in order to understand the majority of the subjects discussed in this book.

The "Base" of a Number

In mathematics, a base is the number of symbols that are used to count with. Usually, we use base 10. The symbols used are the digits from 0 to 9. If we want to count in base 4, then we would use only 4 symbols (0, 1, 2, and 3, for example).

As we count in base 10 we proceed as follows:

- We begin with zero (0);
- To go to the next number we replace the right-most digit with the next symbol in the series (0 becomes 1, 1 becomes 2, etc.);
- When the right-most digit is the last in the series (9), we replace it with the first (0) and we replace the digit to the left with the next symbol in the series (if there is no digit to the left, we say that it was 0).

This general principle is the same in all bases, the only dif ference being the symbol list used.

For example, to count in base 4, we would have: 0_4, 1_4, 2_4, 3_4, 10_4, 11_4, 12_4, 13_4, 20_4, 21_4, 22_4, 23_4, 30_4, 31_4, 32_4, 33_4, 100_4, 101_4,... (which, in base 10, corresponds to the sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1 1, 12, 13, 14, 15, 16, 17,...).

Note, however, that the number 102_4 would read "one-zero-two"—*not* "one hundred two," which is our common lingual notation that can only be used with base-10 numbers.

Two bases are frequently used with computers: base 2, which is called binary, and base 16, which is called hexadecimal.

Binary

To examine the contents of a memory location, the computer checks for electric current: either there is current present, or there is not. Thus, an electronic computer can only have two basic memory states, 1 or 0. And since only two states are possible, all of computer science is based on calculations in base 2. Such calculations are called boolean algebra, named after George Boole who developed this type of two-state arithmetic in 1846. In base 2, we count as follows: 0, 1, 10, 1 1, 100, 101, 110, 111, 1000,... This idea leads to another: the bit.

The Bit

A bit is a binary unit which can be 0 or 1, and thus corresponds to the basic unit found in computers. These bits are usually grouped together, sometimes by four (to form a nibble), but more often by eight (to form a byte). Note that, in groups, *the order of the bits is important.*

The Nibble

The HP48 groups the bits in blocks of four. These blocks are called nibbles. There are 16 possible nibble values: 0000, 0001, 0010, 001 1, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111.

The Byte

Other computers usually use blocks of 8 bits, or bytes. There are 256 possible byte combinations: 00000000, 00000001, 00000010, ... 1 1111110, 11111111. As you can see, binary is not real great to work with, since you must frequently manipulate very long numbers. A base with more symbols would be much more convenient. If the basic unit is binary, then it would be best to use a base that is a multiple of 2. Hexadecimal, or base 16, is what has been chosen.

Hexadecimal

Hexadecimal, or base 16, needs 16 symbols to count with. There are not enough of the traditional digits, so we add 6 more: A, B, C, D, E, and F. (Of course, the symbols used are not important in and of themselves; you can choose any symbols that you wish to do your mathematics. For example, the symbols { 6, e, and $ } could be used for a base 3 system. You would be able to count, and do mathematics using the sequence of numbers: 6, e, $, e6, ee, e$, $6, $e, $$, e66, e6e, e6$, ee6, eee, ee$,... This might be very clear to you, but others may not completely understand. This is why it would probably be best to use the same symbols as the rest of the world.) With the digits chosen for base 16, we count as follows: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, ...19, 1A, 1B, 1C,...

A nibble can therefore have a value of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, or F. And a byte can have a value of 00, 01, 02, 03, 04, ... 0A, 0B, 0C, 0D, 0E, 0F, 10, ... FE, or FF. As you can see, these numbers are much easier to use than those composed of only zeros and ones.

Converting Between Bases

The following program will produce a table of conversions between binary , decimal, and hexadecimal, for the numbers from 0 to 255, which are the most useful to programmers. Each line will have, in this order , binary, decimal, then hexadecimal, all equal to the same number .

```
CONV (# A709h)
  «
    "■" 0 255
    FOR X
      X 1 DISP X R→B SWAP BIN OVER →STR 3 OVER
      SIZE 1 - SUB "          " SWAP + DUP SIZE
      7 - 999 SUB + DEC OVER →STR 3 OVER SIZE
      1 - SUB "   " SWAP + DUP SIZE 3 - 999
      SUB + HEX SWAP →STR 3 OVER SIZE 1 - SUB
      "   " SWAP + DUP SIZE 2 - 999 SUB + "■" +
    NEXT
  »
```

C. RPL Commands

Here is the complete list of HP 48 RPL commands, listed in alphabetical order (which is the same order in the HP reference manual). This list is divided into the two library parts (#002h and #700h). Note that some commands have no name, perform no function, and are probably reserved by HP for future use.

Each line consists of the name of the function, its command number in hexadecimal, its command number in decimal, then the command address (which can be called with a SYSEVAL). For example, ABS is command #03Dh (61) and can be called by #1AA1Fh SYSEVAL.

These addresses can be used in program objects. For example, to duplicate the object in level 1 three times, using the instructions DUP and DUP2, note from the table that a program object has prologue #02D9Dh and epilog #0312Bh. The desired object is therefore:

"D9D2078BF12ABF1B2130"

This program saves 10 nibbles over the regular method of using the two delimiters (« and »), and still performs exactly the same function.

These tables are also useful to the user that would like to disassemble a particular RPL command (these addresses are addresses of machine language routines in ROM).

The second list of HP48 RPL commands is ordered by command number. Each command is defined by its library number and its command number .

Note that, just as there are commands with the same name in the first list, (commands with the same name, but defined by their context—such as « which can be the beginning of a program or the beginning of local variable assignments with →), there are commands in the second list with the same number. This can be explained by the fact that some "commands," such as DIR, C$, etc., are not real commands. They all have the same function—to serve as delimiters for objects.

The first alphabetized table is for library #002h:

ABS	#03Dh	61	#1AA1Fh	COLCT	#14Dh	333	#20A15h
ACK	#015h	21	#1987Eh	COLΣ	#138h	312	#2009Ah
ACKALL	#014h	20	#19863h	COMB	#081h	129	#1C1F6h
ACOS	#058h	88	#1B72Fh	CON	#0ADh	173	#1D186h
ACOSH	#05Bh	91	#1B830h	CONIC	#0DDh	221	#1E681h
ALOG	#060h	96	#1BA3Dh	CONJ	#03Eh	62	#1AA6Eh
AND	#0E5h	229	#1E783h	CONT	#03Ah	58	#1A888h
APPLY	#102h	258	#1F55Dh	CONVERT	#00Bh	11	#196DBh
APPLY	#103h	259	#1F55Ch	CORR	#121h	289	#1FDC1h
ARC	#0D8h	216	#1E5D2h	COS	#052h	82	#1B505h
ARCHIVE	#160h	352	#2125Ah	COSH	#055h	85	#1B606h
ARG	#04Dh	77	#1B20Bh	COV	#122h	290	#1FDDCh
ARRY→	#0ABh	171	#1D092h	CR	#0F3h	243	#1EEA4h
→ARRY	#0AAh	170	#1D009h	CRDIR	#020h	32	#1A105h
ASIN	#057h	87	#1B6A4h	CROSS	#07Ah	122	#1C01Eh
ASINH	#05Ah	90	#1B7EBh	DATE	#011h	17	#19812h
ASN	#17Bh	379	#224F4h	→DATE	#016h	22	#1989Eh
ASR	#000h	0	#1957Bh	DATE+	#01Fh	31	#199D2h
ATAN	#059h	89	#1B79Ch	D→R	#070h	112	#1BEC8h
ATANH	#05Ch	92	#1B8A2h	DDAYS	#01Eh	30	#199B2h
ATTACH	#165h	357	#21440h	DEC	#091h	145	#1C574h
AUTO	#0C0h	192	#1E1ABh	DECR	#14Ch	332	#209AAh
AXES	#0BAh	186	#1E0BEh	DEFINE	#156h	342	#20D65h
BAR	#0E3h	227	#1E741h	DEG	#087h	135	#1C399h
BARPLOT	#13Ch	316	#20133h	DELALARM	#01Ch	28	#19972h
BAUD	#172h	370	#2200Ch	DELAY	#0F5h	245	#1EF43h
B→R	#00Ah	10	#1968Bh	DELKEYS	#17Dh	381	#22540h
BEEP	#034h	52	#1A5C4h	DEPND	#0C4h	196	#1E22Bh
BESTFIT	#143h	323	#2025Eh	DEPTH	#114h	276	#1FC44h
BIN	#090h	144	#1C559h	DET	#078h	120	#1BFDEh
BINS	#13Bh	315	#2010Eh	DETACH	#166h	358	#2147Ch
BLANK	#0D1h	209	#1E416h	DISP	#032h	50	#1A584h
BOX	#0D0h	208	#1E3ECh	DOERR	#02Ah	42	#1A339h
BUFLEN	#176h	374	#22087h	DOT	#079h	121	#1BFFEh
BYTES	#026h	38	#1A1D9h	DRAW	#0BFh	191	#1E190h
C→PX	#0C7h	199	#1E29Ah	DRAX	#0C1h	193	#1E1C6h
C→R	#09Fh	159	#1C98Eh	DROP	#110h	272	#1FBD8h
CEIL	#068h	104	#1BC0Fh	DROP2	#111h	273	#1FBF3h
CENTR	#0BBh	187	#1E0E8h	DROPN	#115h	277	#1FC64h
CF	#084h	132	#1C205h	DTAG	#180h	384	#22633h
%CH	#07Eh	126	#1C149h	DUP	#10Dh	269	#1FB87h
CHR	#0A5h	165	#1CB66h	DUP2	#10Eh	270	#1FBA2h
CKSM	#171h	369	#21FECh	DUPN	#116h	278	#1FC7Fh
CLEAR	#11Ah	282	#1FCEBh	ENG	#08Ch	140	#1C452h
CLKADJ	#018h	24	#1980Eh	EQ→	#0A8h	168	#1CEE3h
CLLCD	#038h	56	#1A858h	ERASE	#0C5h	197	#1E25Fh
CLOSEIO	#16Ah	362	#21ED5h	ERR0	#02Bh	43	#1A36Dh
CLΣ	#11Ch	284	#1FD2Bh	ERRM	#02Dh	45	#1A3A3h
CLUSR	#15Ah	346	#210FCh	ERRN	#02Ch	44	#1A388h
CLVAR	#15Ah	346	#210FCh	EVAL	#02Eh	46	#1A3BEh
CNRM	#077h	119	#1BFBEh	EXP	#05Dh	93	#1B905h

EXPAN	#14Eh	334	#20A49h
EXPFIT	#141h	321	#201FBh
EXPM	#062h	98	#1BAC2h
e	#042h	66	#1AB23h
FACT	#064h	100	#1BB41h
FC?	#086h	134	#1C360h
FC?C	#08Fh	143	#1C520h
FINDALARM	#01Bh	27	#19940h
FINISH	#16Fh	367	#21FB6h
FIX	#08Ah	138	#1C3EAh
FLOOR	#067h	103	#1BBD9h
FP	#066h	102	#1BBA3h
FREE	#163h	355	#213D1h
FREEZE	#033h	51	#1A5A4h
FS?	#085h	133	#1C313h
FS?C	#08Eh	142	#1C4A1h
FUNCTION	#0DCh	220	#1E661h
GET	#0B2h	178	#1D7C6h
GETI	#0B3h	179	#1D8C7h
GOR	#0D3h	211	#1E456h
GRAD	#089h	137	#1C3CFh
GRAPH	#0C8h	200	#1E2BAh
→GROB	#0D7h	215	#1E5ADh
GXOR	#0D4h	212	#1E4E4h
*H	#0BDh	189	#1E150h
HEX	#092h	146	#1C58Fh
HISTOGRAM	#0E2h	226	#1E721h
HISTPLOT	#13Dh	317	#20167h
HMS+	#074h	116	#1BF5Eh
HMS-	#075h	117	#1BF7Eh
HMS→	#073h	115	#1BF3Eh
→HMS	#072h	114	#1BF1Eh
HOME	#022h	34	#1A140h
IDN	#0AEh	174	#1D2DCh
IFT	#030h	48	#1A4CDh
IFTE	#02Fh	47	#1A3FEh
IM	#09Bh	155	#1C819h
INCR	#14Bh	331	#208F4h
INDEP	#0B7h	183	#1E04Ah
INPUT	#17Ah	378	#224CAh
INV	#04Ch	76	#1B270h
IP	#065h	101	#1BB6Dh
ISOL	#150h	336	#20A93h
i	#043h	67	#1AB45h
KERRM	#175h	373	#2206Ch
KEY	#039h	57	#1A873h
KGET	#16Ch	364	#21F24h
KILL	#028h	40	#1A303h
LABEL	#0C9h	201	#1E2D5h
LAST	#036h	54	#1A604h
LASTARG	#036h	54	#1A604h
LCD→	#0D5h	213	#1E572h
→LCD	#0D6h	214	#1E58Dh
LIBS	#164h	356	#2142Dh
LINE	#0CEh	206	#1E398h
ΣLINE	#13Ah	314	#200F3h
LINFIT	#13Fh	319	#201B1h
LIST→	#09Eh	158	#1C95Ah
→LIST	#098h	152	#1C783h
LN	#05Eh	94	#1B94Fh
LNP1	#061h	97	#1BA8Ch
LOG	#05Fh	95	#1B9C6h
LOGFIT	#140h	320	#201D6h
LR	#12Eh	302	#1FF20h
MANT	#06Fh	111	#1BE9Ch
↑MATCH	#109h	265	#1FA59h
↓MATCH	#10Ah	266	#1FA8Dh
MAX	#06Ah	106	#1BC71h
MAXΣ	#128h	296	#1FE7Eh
MAXR	#040h	64	#1AADFh
MEAN	#129h	297	#1FE99h
MEM	#158h	344	#20FAAh
MENU	#15Ch	348	#21196h
MERGE	#162h	354	#2137Fh
MIN	#06Bh	107	#1BCE3h
MINΣ	#12Ah	298	#1FEB4h
MINR	#041h	65	#1AB01h
MOD	#06Eh	110	#1BE4Dh
NEG	#03Ch	60	#1A995h
NEWOB	#027h	39	#1A2BCh
NOT	#0E7h	231	#1EB8Fh
NΣ	#120h	288	#1FDA6h
NUM	#0A4h	164	#1CB46h
→NUM	#035h	53	#1A5E4h
OBJ→	#0A9h	169	#1CF7Bh
OCT	#093h	147	#1C5AAh
OFF	#029h	41	#1A31Eh
OLDPRT	#0EFh	239	#1EE38h
OPENIO	#169h	361	#21EB5h
OR	#0E6h	230	#1EB09h
ORDER	#159h	345	#20FD9h
OVER	#113h	275	#1FC29h
PARAMETRIC	#0DFh	223	#1E6C1h
PARITY	#173h	371	#2202Ch
PATH	#021h	33	#1A125h
PDIM	#0C3h	195	#1E201h
PERM	#082h	130	#1C236h
PGDIR	#15Fh	351	#2123Ah
PICK	#117h	279	#1FC9Ah
PICT	#0D2h	210	#1E436h
PIX?	#0CDh	205	#1E36Eh
PIXOFF	#0CCh	204	#1E344h
PIXON	#0CBh	203	#1E31Ah
PKT	#179h	377	#2200Dh
PMAX	#0B9h	185	#1E09Eh
PMIN	#0B8h	184	#1E07Eh

POLAR	#0DEh	222	#1E6A1h
POS	#0A1h	161	#1CAB4h
PR1	#0F0h	240	#1EE53h
PREDV	#12Fh	303	#1FF7Ah
PREDX	#131h	305	#1FFBAh
PREDY	#130h	304	#1FF9Ah
PRLCD	#0F6h	246	#1EF63h
PRST	#0F2h	242	#1EE89h
PRSTC	#0F1h	241	#1EE6Eh
PRVAR	#0F4h	244	#1EEBFh
PURGE	#157h	343	#20EFEh
PUT	#0B0h	176	#1D407h
PUTI	#0B1h	177	#1D50Fh
PVARS	#15Eh	350	#211FCh
PVIEW	#0CAh	202	#1E2F0h
PWRFIT	#142h	322	#20220h
PX→C	#0C6h	198	#1E27Ah
→Q	#107h	263	#1F9C4h
→Qπ	#108h	264	#1F9E9h
QUAD	#151h	337	#20A83h
QUOTE	#101h	257	#1F500h
RAD	#088h	136	#1C384h
RAND	#07Fh	127	#1C1B9h
RATIO	#10Ch	268	#1FB5Dh
R→B	#009h	9	#1969Bh
R→C	#099h	153	#1C79Eh
R→D	#071h	113	#1BEF4h
RCEQ	#0F9h	249	#1F133h
RCL	#154h	340	#20B40h
RCLALARM	#01Ah	26	#19928h
RCLF	#096h	150	#1C619h
RCLKEYS	#17Eh	382	#22586h
RCLMENU	#15Dh	349	#211E1h
RCLΣ	#11Dh	285	#1FD46h
RCWS	#095h	149	#1C5FEh
RDM	#0ACh	172	#1D0DFh
RDZ	#080h	128	#1C1D4h
RE	#09Ah	154	#1C7CAh
RECN	#16Dh	365	#21F62h
RECV	#16Eh	366	#21F96h
REPL	#09Dh	157	#1C8EAh
RES	#0BCh	188	#1E126h
RESTORE	#161h	353	#2133Ch
RL	#001h	1	#1959Bh
RLB	#002h	2	#195BBh
RND	#06Ch	108	#1BD55h
RNRM	#076h	118	#1BF9Eh
ROLL	#118h	280	#1FCB5h
ROLLD	#119h	281	#1FCD0h
ROOT	#0FBh	251	#1F16Eh
ROT	#112h	274	#1FC0Eh
RR	#003h	3	#195DBh
RRB	#004h	4	#195FBh
RSD	#07Bh	123	#1C03Eh
RULES	#14Fh	335	#20A7Dh
SAME	#0E4h	228	#1E761h
SBRK	#178h	376	#220C2h
SCALE	#0C2h	194	#1E1E1h
SCATRPLOT	#13Eh	318	#2018Ch
SCATTER	#0E1h	225	#1E701h
SCI	#08Bh	139	#1C41Eh
SCLΣ	#139h	313	#200C4h
SCONJ	#146h	326	#203CCh
SDEV	#12Bh	299	#1FECFh
SEND	#16Bh	363	#21EF0h
SERVER	#170h	368	#21FD1h
SF	#083h	131	#1C274h
SHOW	#152h	338	#20AD3h
SIGN	#04Eh	78	#1B32Ah
SIN	#051h	81	#1B4ACh
SINH	#054h	84	#1B5B7h
SINV	#144h	324	#202CEh
SIZE	#0A0h	160	#1C9B8h
SL	#005h	5	#1961Bh
SLB	#006h	6	#1963Bh
SNEG	#145h	325	#2034Dh
SQ	#050h	80	#1B426h
SR	#007h	7	#1965Bh
SRB	#008h	8	#1967Bh
SRECV	#168h	360	#21E95h
STD	#08Dh	141	#1C486h
STEQ	#0FAh	250	#1F14Eh
STIME	#177h	375	#220A2h
STO	#155h	341	#20CCDh
STO*	#14Ah	330	#20753h
STO+	#147h	327	#2044Bh
STO-	#148h	328	#2053Bh
STO/	#149h	329	#2060Ch
STOALARM	#019h	25	#198FEh
STOF	#097h	151	#1C67Fh
STOKEYS	#17Ch	380	#22514h
STOΣ	#11Bh	283	#1FD0Bh
STR→	#0A3h	163	#1CB26h
→STR	#0A2h	162	#1CB08h
STWS	#094h	148	#1C5C5h
SUB	#09Ch	156	#1C85Ch
SWAP	#10Fh	271	#1FBBDh
SYSEVAL	#031h	49	#1A52Eh
%T	#07Dh	125	#1C0D7h
→TAG	#17Fh	383	#225BEh
TAN	#053h	83	#1B55Eh
TANH	#056h	86	#1B655h
TAYLR	#153h	339	#20B20h
TEXT	#0D9h	217	#1E606h
TICKS	#012h	18	#1982Dh
TIME	#010h	16	#197F7h

→TIME	#017h	23	#1988Eh
TLINE	#0CFh	207	#1E3C2h
TMENU	#15Bh	347	#2115Dh
TOT	#12Ch	300	#1FEEAh
TRANSIO	#174h	372	#2204Ch
TRN	#0AFh	175	#1D392h
TRNC	#06Dh	109	#1BD01h
TRUTH	#0E0h	224	#1E6E1h
TSTR	#01Dh	29	#19992h
TVARS	#025h	37	#1A1AFh
TYPE	#0A6h	166	#1CB86h
UBASE	#00Eh	14	#19771h
UFACT	#00Fh	15	#197A5h
→UNIT	#00Dh	13	#1974Fh
UPDIR	#023h	35	#1A15Bh
UTPC	#134h	308	#2001Ah
UTPF	#136h	310	#2005Ah
UTPN	#135h	309	#2003Ah
UTPT	#137h	311	#2007Ah
UVAL	#00Ch	12	#1971Bh
VAR	#12Dh	301	#1FF05h
VARS	#024h	36	#1A194h
V→	#0B4h	180	#1DD06h
→V2	#0B5h	181	#1DE66h
→V3	#0B6h	182	#1DEC2h
VTYPE	#0A7h	167	#1CE28h
*W	#0BEh	190	#1E170h
WAIT	#037h	55	#1A71Fh
WSLOG	#013h	19	#19840h
ΣX	#123h	291	#1FDF7h
ΣX^2	#125h	293	#1FE2Dh
XCOL	#132h	306	#1FFDAh
XMIT	#167h	359	#21E75h
XOR	#0E8h	232	#1E8F6h
XPON	#069h	105	#1BC45h
XRNG	#0DAh	218	#1E621h
XROOT	#04Ah	74	#1B185h
ΣX*Y	#127h	295	#1FE63h
ΣY	#124h	292	#1FE12h
ΣY^2	#126h	294	#1FE40h
YCOL	#133h	307	#1FFFAh
YRNG	#0DBh	219	#1E641h
+	#044h	68	#1AB67h
+	#045h	69	#1ACDDh
-	#046h	70	#1AD09h
*	#047h	71	#1ADEEh
/	#048h	72	#1AF05h
^	#049h	73	#1B02Dh
<	#0EBh	235	#1EB8Eh
≤	#0EDh	237	#1ECFCh
>	#0ECh	236	#1EC5Dh
≥	#0EEh	238	#1ED98h
=	#03Bh	59	#1A8D8h
==	#0E9h	233	#1E972h
≠	#0EAh	234	#1EA9Dh
!	#063h	99	#1BB02h
∫	#0FCh	252	#1F1D4h
∫	#0FDh	253	#1F223h
∂	#0F7h	247	#1EF7Eh
∂	#0F8h	248	#1EFD2h
%	#07Ch	124	#1C060h
π	#03Fh	63	#1AABDh
Σ	#0FEh	254	#1F2C9h
Σ+	#11Eh	286	#1FD61h
Σ-	#11Fh	287	#1FD8Bh
√	#04Fh	79	#1B374h
|	#0FFh	255	#1F354h
|	#100h	256	#1F3F3h
_	#10Bh	267	#1FAEBh
	#104h	260	#1F640h
	#105h	261	#1F996h
	#106h	262	#1F9AEh

Alphabetized for library #700h:

C#	#01Bh	27	#23813h
CASE	#019h	25	#2378Dh
DIR	#01Bh	27	#23813h
DO	#007h	7	#230C3h
ELSE	#002h	2	#22FB5h
END	#003h	3	#22FD5h
END	#016h	22	#23694h
END	#017h	23	#236B9h
FOR	#00Ah	10	#231A0h
GROB	#01Bh	27	#23813h
HALT	#00Eh	14	#23472h
IF	#000h	0	#22EC3h
IFERR	#00Dh	13	#2330Fh
NEXT	#00Bh	11	#2324Ch
PROMPT	#01Ch	28	#23824h
REPEAT	#006h	6	#2305Dh
START	#009h	9	#23103h
STEP	#00Ch	12	#23380h
THEN	#001h	1	#22EFAh
THEN	#018h	24	#2371Fh
THEN	#01Ah	26	#237A8h
UNTIL	#008h	8	#230EDh
WHILE	#005h	5	#23033h
XLIB	#01Bh	27	#23813h
«	#012h	18	#2361Eh
»	#011h	17	#235FEh
»	#013h	19	#23639h
→	#004h	4	#22FEBh
→	#010h	16	#234C1h
'	#014h	20	#23654h
'	#015h	21	#23679h
	#00Fh	17	#2349Ch

The numerical table for library #002h:

	#000h	#1957Bh	ASR
	#001h	#1959Bh	RL
	#002h	#195BBh	RLB
	#003h	#195DBh	RR
	#004h	#195FBh	RRB
	#005h	#1961Bh	SL
	#006h	#1963Bh	SLB
	#007h	#1965Bh	SR
	#008h	#1967Bh	SRB
	#009h	#1969Bh	R→B
10	#00Ah	#196BBh	B→R
11	#00Bh	#196DBh	CONVERT
12	#00Ch	#1971Bh	UVAL
13	#00Dh	#1974Fh	→UNIT
14	#00Eh	#19771h	UBASE
15	#00Fh	#197A5h	UFACT
16	#010h	#197F7h	TIME
17	#011h	#19812h	DATE
18	#012h	#1982Dh	TICKS
19	#013h	#19848h	WSLOG
20	#014h	#19863h	ACKALL
21	#015h	#1987Eh	ACK
22	#016h	#1989Eh	→DATE
23	#017h	#198BEh	→TIME
24	#018h	#198DEh	CLKADJ
25	#019h	#198FEh	STOALARM
26	#01Ah	#19928h	RCLALARM
27	#01Bh	#19948h	FINDALARM
28	#01Ch	#19972h	DELALARM
29	#01Dh	#19992h	TSTR
30	#01Eh	#199B2h	DDAYS
31	#01Fh	#199D2h	DATE+
32	#020h	#1A105h	CRDIR
33	#021h	#1A125h	PATH
34	#022h	#1A140h	HOME
35	#023h	#1A15Bh	UPDIR
36	#024h	#1A194h	VARS
37	#025h	#1A1AFh	TVARS
38	#026h	#1A1D9h	BYTES
39	#027h	#1A2BCh	NEWOB
40	#028h	#1A303h	KILL
41	#029h	#1A31Eh	OFF
42	#02Ah	#1A339h	DOERR
43	#02Bh	#1A36Dh	ERR0
44	#02Ch	#1A388h	ERRN
45	#02Dh	#1A3A3h	ERRM
46	#02Eh	#1A3BEh	EVAL
47	#02Fh	#1A3FEh	IFTE
48	#030h	#1A4CDh	IFT
49	#031h	#1A52Eh	SYSEVAL
50	#032h	#1A584h	DISP
51	#033h	#1A5A4h	FREEZE
52	#034h	#1A5C4h	BEEP
53	#035h	#1A5E4h	→NUM
54	#036h	#1A604h	LAST
54	#036h	#1A604h	LASTARG
55	#037h	#1A71Fh	WAIT
56	#038h	#1A85Bh	CLLCD
57	#039h	#1A873h	KEY
58	#03Ah	#1A88Bh	CONT
59	#03Bh	#1A8DBh	=
60	#03Ch	#1A995h	NEG
61	#03Dh	#1AA1Fh	ABS
62	#03Eh	#1AA6Eh	CONJ
63	#03Fh	#1AABDh	π
64	#040h	#1AADFh	MAXR
65	#041h	#1AB01h	MINR
66	#042h	#1AB23h	e
67	#043h	#1AB45h	i
68	#044h	#1AB67h	+
69	#045h	#1ACDDh	+
70	#046h	#1AD09h	-
71	#047h	#1ADEEh	*
72	#048h	#1AF05h	/
73	#049h	#1B02Dh	^
74	#04Ah	#1B185h	XROOT
76	#04Ch	#1B278h	INV
77	#04Dh	#1B2DBh	ARG
78	#04Eh	#1B32Ah	SIGN
79	#04Fh	#1B374h	√
80	#050h	#1B426h	SQ
81	#051h	#1B4ACh	SIN
82	#052h	#1B505h	COS
83	#053h	#1B55Eh	TAN
84	#054h	#1B5B7h	SINH
85	#055h	#1B606h	COSH
86	#056h	#1B655h	TANH
87	#057h	#1B6A4h	ASIN
88	#058h	#1B72Fh	ACOS
89	#059h	#1B79Ch	ATAN
90	#05Ah	#1B7EBh	ASINH
91	#05Bh	#1B830h	ACOSH
92	#05Ch	#1B8A2h	ATANH
93	#05Dh	#1B905h	EXP
94	#05Eh	#1B94Fh	LN
95	#05Fh	#1B9C6h	LOG
96	#060h	#1BA3Dh	ALOG
97	#061h	#1BA8Ch	LNP1
98	#062h	#1BAC2h	EXPM
99	#063h	#1BB02h	!
100	#064h	#1BB41h	FACT
101	#065h	#1BB6Dh	IP

102	#066h	#1BBA3h	FP
103	#067h	#1BBD9h	FLOOR
104	#068h	#1BC0Fh	CEIL
105	#069h	#1BC45h	XPON
106	#06Ah	#1BC71h	MAX
107	#06Bh	#1BCE3h	MIN
108	#06Ch	#1BD55h	RND
109	#06Dh	#1BDD1h	TRNC
110	#06Eh	#1BE4Dh	MOD
111	#06Fh	#1BE9Ch	MANT
112	#070h	#1BEC8h	D→R
113	#071h	#1BEF4h	R→D
114	#072h	#1BF1Eh	→HMS
115	#073h	#1BF3Eh	HMS→
116	#074h	#1BF5Eh	HMS+
117	#075h	#1BF7Eh	HMS-
118	#076h	#1BF9Eh	RNRM
119	#077h	#1BFBEh	CNRM
120	#078h	#1BFDEh	DET
121	#079h	#1BFFEh	DOT
122	#07Ah	#1C01Eh	CROSS
123	#07Bh	#1C03Eh	RSD
124	#07Ch	#1C060h	%
125	#07Dh	#1C0D7h	%T
126	#07Eh	#1C149h	%CH
127	#07Fh	#1C1B9h	RAND
128	#080h	#1C1D4h	RDZ
129	#081h	#1C1F6h	COMB
130	#082h	#1C236h	PERM
131	#083h	#1C274h	SF
132	#084h	#1C2D5h	CF
133	#085h	#1C313h	FS?
134	#086h	#1C360h	FC?
135	#087h	#1C399h	DEG
136	#088h	#1C3B4h	RAD
137	#089h	#1C3CFh	GRAD
138	#08Ah	#1C3EAh	FIX
139	#08Bh	#1C41Eh	SCI
140	#08Ch	#1C452h	ENG
141	#08Dh	#1C4B6h	STD
142	#08Eh	#1C4A1h	FS?C
143	#08Fh	#1C520h	FC?C
144	#090h	#1C559h	BIN
145	#091h	#1C574h	DEC
146	#092h	#1C58Fh	HEX
147	#093h	#1C5AAh	OCT
148	#094h	#1C5C5h	STWS
149	#095h	#1C5FEh	RCWS
150	#096h	#1C619h	RCLF
151	#097h	#1C67Fh	STOF
152	#098h	#1C783h	→LIST
153	#099h	#1C79Eh	R→C
154	#09Ah	#1C7CAh	RE
155	#09Bh	#1C819h	IM
156	#09Ch	#1C85Ch	SUB
157	#09Dh	#1C8EAh	REPL
158	#09Eh	#1C95Ah	LIST→
159	#09Fh	#1C98Eh	C→R
160	#0A0h	#1C9B8h	SIZE
161	#0A1h	#1CAB4h	POS
162	#0A2h	#1CB0Bh	→STR
163	#0A3h	#1CB26h	STR→
164	#0A4h	#1CB46h	NUM
165	#0A5h	#1CB66h	CHR
166	#0A6h	#1CB86h	TYPE
167	#0A7h	#1CE28h	VTYPE
168	#0A8h	#1CEE3h	EQ→
169	#0A9h	#1CF78h	OBJ→
170	#0AAh	#1D009h	→ARRY
171	#0ABh	#1D092h	ARRY→
172	#0ACh	#1D0DFh	RDM
173	#0ADh	#1D186h	CON
174	#0AEh	#1D20Ch	IDN
175	#0AFh	#1D392h	TRN
176	#0B0h	#1D407h	PUT
177	#0B1h	#1D5DFh	PUTI
178	#0B2h	#1D7C6h	GET
179	#0B3h	#1D8C7h	GETI
180	#0B4h	#1DD06h	V→
181	#0B5h	#1DE66h	→V2
182	#0B6h	#1DEC2h	→V3
183	#0B7h	#1E04Ah	INDEP
184	#0B8h	#1E07Eh	PMIN
185	#0B9h	#1E09Eh	PMAX
186	#0BAh	#1E0BEh	AXES
187	#0BBh	#1E0E8h	CENTR
188	#0BCh	#1E126h	RES
189	#0BDh	#1E150h	*H
190	#0BEh	#1E170h	*W
191	#0BFh	#1E190h	DRAW
192	#0C0h	#1E1ABh	AUTO
193	#0C1h	#1E1C6h	DRAX
194	#0C2h	#1E1E1h	SCALE
195	#0C3h	#1E201h	PDIM
196	#0C4h	#1E22Bh	DEPND
197	#0C5h	#1E25Fh	ERASE
198	#0C6h	#1E27Ah	PX→C
199	#0C7h	#1E29Ah	C→PX
200	#0C8h	#1E2BAh	GRAPH
201	#0C9h	#1E2D5h	LABEL
202	#0CAh	#1E2F0h	PVIEW
203	#0CBh	#1E31Ah	PIXON
204	#0CCh	#1E344h	PIXOFF
205	#0CDh	#1E36Eh	PIX?
206	#0CEh	#1E398h	LINE
207	#0CFh	#1E3C2h	TLINE

208	#0D0h	#1E3ECh	BOX
209	#0D1h	#1E416h	BLANK
210	#0D2h	#1E436h	PICT
211	#0D3h	#1E456h	GOR
212	#0D4h	#1E4E4h	GXOR
213	#0D5h	#1E572h	LCD→
214	#0D6h	#1E580h	→LCD
215	#0D7h	#1E5A0h	→GROB
216	#0D8h	#1E5D2h	ARC
217	#0D9h	#1E606h	TEXT
218	#0DAh	#1E621h	XRNG
219	#0DBh	#1E641h	YRNG
220	#0DCh	#1E661h	FUNCTION
221	#0DDh	#1E681h	CONIC
222	#0DEh	#1E6A1h	POLAR
223	#0DFh	#1E6C1h	PARAMETRIC
224	#0E0h	#1E6E1h	TRUTH
225	#0E1h	#1E701h	SCATTER
226	#0E2h	#1E721h	HISTOGRAM
227	#0E3h	#1E741h	BAR
228	#0E4h	#1E761h	SAME
229	#0E5h	#1E783h	AND
230	#0E6h	#1E809h	OR
231	#0E7h	#1E88Fh	NOT
232	#0E8h	#1E8F6h	XOR
233	#0E9h	#1E972h	==
234	#0EAh	#1EA90h	≠
235	#0EBh	#1EB8Eh	<
236	#0ECh	#1EC5Dh	>
237	#0EDh	#1ECFCh	≤
238	#0EEh	#1ED98h	≥
239	#0EFh	#1EE38h	OLDPRT
240	#0F0h	#1EE53h	PR1
241	#0F1h	#1EE6Eh	PRSTC
242	#0F2h	#1EE89h	PRST
243	#0F3h	#1EEA4h	CR
244	#0F4h	#1EEBFh	PRVAR
245	#0F5h	#1EF43h	DELAY
246	#0F6h	#1EF63h	PRLCD
247	#0F7h	#1EF7Eh	∂
248	#0F8h	#1EFD2h	∂
249	#0F9h	#1F133h	RCEQ
250	#0FAh	#1F14Eh	STEQ
251	#0FBh	#1F16Eh	ROOT
252	#0FCh	#1F1D4h	∫
253	#0FDh	#1F223h	∫
254	#0FEh	#1F2C9h	Σ
255	#0FFh	#1F354h	|
256	#100h	#1F3F3h	|
257	#101h	#1F500h	QUOTE
258	#102h	#1F55Dh	APPLY
259	#103h	#1F55Ch	APPLY
260	#104h	#1F640h	
261	#105h	#1F996h	
262	#106h	#1F9AEh	
263	#107h	#1F9C4h	→Q
264	#108h	#1F9E9h	→Qπ
265	#109h	#1FA59h	↑MATCH
266	#10Ah	#1FA8Dh	↓MATCH
267	#10Bh	#1FAEBh	_
268	#10Ch	#1FB5Dh	R̄ATIO
269	#10Dh	#1FB87h	DUP
270	#10Eh	#1FBA2h	DUP2
271	#10Fh	#1FBBDh	SWAP
272	#110h	#1FBD8h	DROP
273	#111h	#1FBF3h	DROP2
274	#112h	#1FC0Eh	ROT
275	#113h	#1FC29h	OVER
276	#114h	#1FC44h	DEPTH
277	#115h	#1FC64h	DROPN
278	#116h	#1FC7Fh	DUPN
279	#117h	#1FC9Ah	PICK
280	#118h	#1FCB5h	ROLL
281	#119h	#1FCD0h	ROLLD
282	#11Ah	#1FCEBh	CLEAR
283	#11Bh	#1FD08h	STOΣ
284	#11Ch	#1FD2Bh	CLΣ
285	#11Dh	#1FD46h	RCLΣ
286	#11Eh	#1FD61h	Σ+
287	#11Fh	#1FD88h	Σ-
288	#120h	#1FDA6h	NΣ
289	#121h	#1FDC1h	CORR
290	#122h	#1FDDCh	COV
291	#123h	#1FDF7h	ΣX
292	#124h	#1FE12h	ΣY
293	#125h	#1FE2Dh	ΣX^2
294	#126h	#1FE48h	ΣY^2
295	#127h	#1FE63h	ΣX*Y
296	#128h	#1FE7Eh	MAXΣ
297	#129h	#1FE99h	MEAN
298	#12Ah	#1FEB4h	MINΣ
299	#12Bh	#1FECFh	SDEV
300	#12Ch	#1FEEAh	TOT
301	#12Dh	#1FF05h	VAR
302	#12Eh	#1FF20h	LR
303	#12Fh	#1FF7Ah	PREDV
304	#130h	#1FF9Ah	PREDY
305	#131h	#1FFBAh	PREDX
306	#132h	#1FFDAh	XCOL
307	#133h	#1FFFAh	YCOL
308	#134h	#2001Ah	UTPC
309	#135h	#2003Ah	UTPN
310	#136h	#2005Ah	UTPF
311	#137h	#2007Ah	UTPT
312	#138h	#2009Ah	COLΣ
313	#139h	#200C4h	SCLΣ

314	#13Ah	#200F3h	ΣLINE
315	#13Bh	#2010Eh	BINS
316	#13Ch	#20133h	BARPLOT
317	#13Dh	#20167h	HISTPLOT
318	#13Eh	#2018Ch	SCATRPLOT
319	#13Fh	#201B1h	LINFIT
320	#140h	#201D6h	LOGFIT
321	#141h	#201FBh	EXPFIT
322	#142h	#20220h	PWRFIT
323	#143h	#2025Eh	BESTFIT
324	#144h	#202CEh	SINV
325	#145h	#2034Dh	SNEG
326	#146h	#203CCh	SCONJ
327	#147h	#2044Bh	STO+
328	#148h	#2053Bh	STO-
329	#149h	#2060Ch	STO/
330	#14Ah	#20753h	STO*
331	#14Bh	#208F4h	INCR
332	#14Ch	#209AAh	DECR
333	#14Dh	#20A15h	COLCT
334	#14Eh	#20A49h	EXPAN
335	#14Fh	#20A7Dh	RULES
336	#150h	#20A93h	ISOL
337	#151h	#20AB3h	QUAD
338	#152h	#20AD3h	SHOW
339	#153h	#20B20h	TAYLR
340	#154h	#20B40h	RCL
341	#155h	#20CCDh	STO
342	#156h	#20D65h	DEFINE
343	#157h	#20EFEh	PURGE
344	#158h	#20FAAh	MEM
345	#159h	#20FD9h	ORDER
346	#15Ah	#210FCh	CLUSR
346	#15Ah	#210FCh	CLVAR
347	#15Bh	#2115Dh	TMENU
348	#15Ch	#21196h	MENU
349	#15Dh	#211E1h	RCLMENU
350	#15Eh	#211FCh	PVARS
351	#15Fh	#2123Ah	PGDIR
352	#160h	#2125Ah	ARCHIVE
353	#161h	#2133Ch	RESTORE
354	#162h	#2137Fh	MERGE
355	#163h	#213D1h	FREE
356	#164h	#2142Dh	LIBS
357	#165h	#21448h	ATTACH
358	#166h	#2147Ch	DETACH
359	#167h	#21E75h	XMIT
360	#168h	#21E95h	SRECV
361	#169h	#21EB5h	OPENIO
362	#16Ah	#21ED5h	CLOSEIO
363	#16Bh	#21EF0h	SEND
364	#16Ch	#21F24h	KGET
365	#16Dh	#21F62h	RECN

366	#16Eh	#21F96h	RECV
367	#16Fh	#21FB6h	FINISH
368	#170h	#21FD1h	SERVER
369	#171h	#21FECh	CKSM
370	#172h	#2200Ch	BAUD
371	#173h	#2202Ch	PARITY
372	#174h	#2204Ch	TRANSIO
373	#175h	#2206Ch	KERRM
374	#176h	#22087h	BUFLEN
375	#177h	#220A2h	STIME
376	#178h	#220C2h	SBRK
377	#179h	#220DDh	PKT
378	#17Ah	#224CAh	INPUT
379	#17Bh	#224F4h	ASN
380	#17Ch	#22514h	STOKEYS
381	#17Dh	#22548h	DELKEYS
382	#17Eh	#22586h	RCLKEYS
383	#17Fh	#225BEh	→TAG
384	#180h	#22633h	DTAG

Numerical table for library #700h:

0	#000h	#22EC3h	IF
1	#001h	#22EFAh	THEN
2	#002h	#22FB5h	ELSE
3	#003h	#22FD5h	END
4	#004h	#22FEBh	→
5	#005h	#23033h	WHILE
6	#006h	#2305Dh	REPEAT
7	#007h	#230C3h	DO
8	#008h	#230EDh	UNTIL
9	#009h	#23103h	START
10	#00Ah	#231A0h	FOR
11	#00Bh	#2324Ch	NEXT
12	#00Ch	#23380h	STEP
13	#00Dh	#2330Fh	IFERR
14	#00Eh	#23472h	HALT
17	#00Fh	#2349Ch	
16	#010h	#234C1h	→
17	#011h	#235FEh	»
18	#012h	#2361Eh	«
19	#013h	#23639h	»
20	#014h	#23654h	'
21	#015h	#23679h	'
22	#016h	#23694h	END
23	#017h	#236B9h	END
24	#018h	#2371Fh	THEN
25	#019h	#2378Dh	CASE
26	#01Ah	#237A8h	THEN
27	#01Bh	#23813h	C#
27	#01Bh	#23813h	DIR
27	#01Bh	#23813h	GROB
27	#01Bh	#23813h	XLIB
28	#01Ch	#23824h	PROMPT

D. Objects in ROM

This is an address list of objects in ROM. This list is not complete, but gives many useful objects. Rather than coding some object that you need, you can simply refer to it with a ROM address. Notice: Addresses greater than #70000h are objects in the hidden ROM and cannot be used directly . You will need to use the ROMRCL routine found in the **Library of Programs**.

System Binaries

#03FEFh	<0h>	0
#03FF9h	<1h>	1
#04003h	<2h>	2
#0400Dh	<3h>	3
#04017h	<4h>	4
#04021h	<5h>	5
#0402Bh	<6h>	6
#04035h	<7h>	7
#0403Fh	<8h>	8
#04049h	<9h>	9
#04053h	<Ah>	10
#0405Dh	<Bh>	11
#04067h	<Ch>	12
#04071h	<Dh>	13
#0407Bh	<Eh>	14
#04085h	<Fh>	15
#0408Fh	<10h>	16
#04099h	<11h>	17
#040A3h	<12h>	18
#040ADh	<13h>	19
#040B7h	<14h>	20
#040C1h	<15h>	21
#040CBh	<16h>	22
#040D5h	<17h>	23
#040DFh	<18h>	24
#040E9h	<19h>	25
#040F3h	<1Ah>	26
#040FDh	<1Bh>	27
#04107h	<1Ch>	28
#04111h	<1Dh>	29
#0411Bh	<1Eh>	30
#04125h	<1Fh>	31
#0412Fh	<20h>	32
#04139h	<21h>	33
#04143h	<22h>	34
#0414Dh	<23h>	35
#04157h	<24h>	36
#04161h	<25h>	37
#0416Bh	<26h>	38
#04175h	<27h>	39
#0417Fh	<28h>	40
#04189h	<29h>	41
#04193h	<2Ah>	42
#0419Dh	<2Bh>	43
#64B12h	<2Ch>	44
#64B1Ch	<2Dh>	45
#64B26h	<2Eh>	46
#64B30h	<2Fh>	47
#64B3Ah	<30h>	48
#64B44h	<31h>	49
#64B4Eh	<32h>	50
#64B58h	<33h>	51
#64B62h	<34h>	52
#64B6Ch	<35h>	53
#64B76h	<36h>	54
#64B80h	<37h>	55
#64B8Ah	<38h>	56
#64B94h	<39h>	57
#64B9Eh	<3Ah>	58
#64BA8h	<3Bh>	59
#64BB2h	<3Ch>	60
#64BBCh	<3Dh>	61
#64BC6h	<3Eh>	62
#64BD0h	<3Fh>	63
#64BDAh	<40h>	64
#64BE4h	<41h>	65
#64BEEh	<42h>	66
#64BF8h	<43h>	67
#64C02h	<44h>	68
#64C0Ch	<45h>	69
#64C16h	<46h>	70
#64C20h	<4Ah>	74
#2D96Ah	<4Bh>	75
#6B5C4h	<4Dh>	77
#64C2Ah	<4Fh>	79
#64C34h	<50h>	80
#64C3Eh	<51h>	81
#64C48h	<52h>	82
#64C52h	<53h>	83
#64C5Ch	<54h>	84
#64C66h	<55h>	85
#64C70h	<56h>	86
#64C7Ah	<57h>	87
#3A215h	<58h>	88
#64C84h	<5Bh>	91
#1CCD8h	<5Fh>	95
#64C8Eh	<60h>	96
#64C98h	<61h>	97
#6B696h	<61h>	97
#64CA2h	<62h>	98
#64CACh	<64h>	100
#64CB6h	<65h>	101

#6D98Ch	<68h>	104
#3A20Bh	<6Eh>	110
#64CC0h	<6Fh>	111
#64CCAh	<70h>	112
#64CD4h	<71h>	113
#64CDEh	<72h>	114
#64CE8h	<73h>	115
#64CF2h	<74h>	116
#64CFCh	<75h>	117
#64D06h	<7Ah>	122
#39980h	<7Dh>	125
#6C947h	<7Fh>	127
#64D10h	<80h>	128
#64D1Ah	<82h>	130
#64D24h	<83h>	131
#46BE3h	<84h>	132
#2F4A2h	<86h>	134
#64D2Eh	<8Fh>	143
#64D38h	<91h>	145
#64D42h	<92h>	146
#6C68Bh	<93h>	147
#64D4Ch	<9Ah>	154
#64D56h	<9Eh>	158
#64D60h	<9Fh>	159
#64D6Ah	<A0h>	160
#64D74h	<A1h>	161
#64D7Eh	<A2h>	162
#64D88h	<A5h>	165
#64D92h	<A6h>	166
#64D9Ch	<A7h>	167
#64DA6h	<A9h>	169
#64DB0h	<AAh>	170
#64DBAh	<AEh>	174
#1CD69h	<AFh>	175
#64DC4h	<B1h>	177
#17F4Ah	<B8h>	184
#64DCEh	<BBh>	187
#64DD8h	<C0h>	192
#2002Ch	<C8h>	200
#64DE2h	<CCh>	204
#64DECh	<D0h>	208
#64DF6h	<E1h>	225
#64E00h	<EAh>	234
#64E0Ah	<EEh>	238
#64E14h	<F0h>	240
#64E1Eh	<FDh>	253
#64E28h	<FFh>	255
#64E32h	<100h>	256
#64E3Ch	<102h>	258
#50E45h	<104h>	260
#64E46h	<106h>	262
#64E50h	<107h>	263
#64E5Ah	<110h>	272
#64E64h	<111h>	273
#15E0Bh	<112h>	274
#15D6Fh	<117h>	279
#15DABh	<118h>	280
#64E6Eh	<123h>	291
#64E78h	<124h>	292
#31C5Eh	<127h>	295
#33CA1h	<12Fh>	303
#64E82h	<131h>	305
#64E8Ch	<132h>	306
#64E96h	<133h>	307
#64EA0h	<134h>	308
#64EAAh	<135h>	309
#64EB4h	<136h>	310
#64EBEh	<137h>	311
#64EC8h	<138h>	312
#64ED2h	<139h>	313
#64EDCh	<13Ah>	314
#64EE6h	<13Bh>	315
#64EF0h	<13Dh>	317
#64EFAh	<13Eh>	318
#64F04h	<151h>	337
#64F0Eh	<200h>	512
#4ECFDh	<201h>	513
#4ED25h	<202h>	514
#2E706h	<204h>	516
#64F18h	<205h>	517
#6737Bh	<206h>	518
#1B147h	<304h>	772
#64F22h	<311h>	785
#1C930h	<313h>	787
#64F2Ch	<411h>	1041
#64F36h	<412h>	1042
#64F40h	<444h>	1092
#64F4Ah	<451h>	1105
#64F54h	<452h>	1106
#37DE7h	<501h>	1281
#472C8h	<502h>	1282
#472D2h	<503h>	1283
#472DCh	<504h>	1284
#472E6h	<505h>	1285
#472F0h	<506h>	1286
#64F5Eh	<510h>	1296
#64F68h	<511h>	1297
#1C93Fh	<515h>	1301
#64F72h	<550h>	1360
#50E4Fh	<602h>	1538
#4A320h	<605h>	1541
#4A32Ah	<606h>	1542
#4A334h	<607h>	1543
#4A33Eh	<608h>	1544
#4A348h	<609h>	1545
#4A352h	<60Ah>	1546
#4A35Ch	<60Bh>	1547
#4A366h	<60Ch>	1548
#4A370h	<60Dh>	1549
#4A37Ah	<60Eh>	1550
#4A384h	<60Fh>	1551
#4A38Eh	<610h>	1552

#4A398h	<611h>	1553	#64FF4h	<A22h>	2594
#4A3A2h	<612h>	1554	#64FFEh	<A2Ah>	2602
#4A3ACh	<613h>	1555	#65008h	<A61h>	2657
#4A3B6h	<614h>	1556	#65012h	<A62h>	2658
#4A3C0h	<615h>	1557	#6501Ch	<A65h>	2661
#4A3CAh	<616h>	1558	#65026h	<A6Eh>	2670
#4A3D4h	<617h>	1559	#65030h	<AA1h>	2721
#4A3DEh	<618h>	1560	#6503Ah	<AA2h>	2722
#4A3E8h	<619h>	1561	#65044h	<AAAh>	2730
#4A3F2h	<61Ah>	1562	#28AE7h	<B01h>	2817
#4A3FCh	<61Bh>	1563	#2D5C3h	<C02h>	3074
#4A406h	<61Ch>	1564	#6504Eh	<C06h>	3078
#4A410h	<61Dh>	1565	#65058h	<C07h>	3079
#4A41Ah	<61Eh>	1566	#65062h	<C08h>	3080
#4A424h	<61Fh>	1567	#6506Ch	<C0Ah>	3082
#4A42Eh	<620h>	1568	#65076h	<C0Bh>	3083
#4A438h	<621h>	1569	#2DA7Ch	<C0Ch>	3084
#4A442h	<622h>	1570	#2F0E0h	<C0Dh>	3085
#4A44Ch	<623h>	1571	#2FBEEh	<C0Eh>	3086
#4A456h	<624h>	1572	#2FA9Ch	<C0Fh>	3087
#4A460h	<628h>	1576	#2F04Ah	<C10h>	3088
#4A46Ah	<629h>	1577	#2D06Fh	<C11h>	3089
#4A474h	<62Ah>	1578	#2EC39h	<C12h>	3090
#4A47Eh	<62Bh>	1579	#3182Ch	<C15h>	3093
#4A488h	<62Ch>	1580	#31BF1h	<C16h>	3094
#4A492h	<62Dh>	1581	#67385h	<C17h>	3095
#4A49Ch	<62Eh>	1582	#1C889h	<C22h>	3106
#20496h	<644h>	1604	#1E50Ch	<C2Ch>	3116
#64F86h	<650h>	1616	#1C87Ah	<C55h>	3157
#64F90h	<700h>	1792	#1E4EEh	<C5Ch>	3164
#1D440h	<710h>	1808	#26289h	<CFFh>	3327
#1D42Fh	<750h>	1872	#65080h	<DFFh>	3583
#08E14h	<7FFh>	2047	#6508Ah	<E00h>	3584
#4B32Dh	<800h>	2048	#1E5DCh	<2111h>	8465
#1C8A7h	<822h>	2082	#03F88h	<2933h>	10547
#1E540h	<82Ch>	2092	#03FDBh	<2955h>	10581
#1C890h	<855h>	2133	#03F95h	<2977h>	10615
#1C912h	<85Ch>	2140	#03F9Fh	<2A74h>	10868
#1E49Ch	<85Ch>	2140	#03FC7h	<2A96h>	10902
#1E52Ah	<85Ch>	2140	#19173h	<2A96h>	10902
#64F9Ah	<861h>	2145	#03FBDh	<2AB8h>	10936
#64FA4h	<862h>	2146	#03FE5h	<2ADAh>	10970
#64FAEh	<865h>	2149	#25C37h	<2B1Eh>	11038
#64FB8h	<86Eh>	2158	#03FB3h	<2D9Dh>	11677
#2D04Ah	<8F1h>	2289	#03FA9h	<2E40h>	11840
#2D03Bh	<9F1h>	2545	#03FD1h	<2E6Dh>	11885
#33CBFh	<A01h>	2561	#3974Dh	<3039h>	12345
#33CD3h	<A02h>	2562	#16AD6h	<4000h>	16384
#34301h	<A03h>	2563	#16AE5h	<5000h>	20480
#33D91h	<A04h>	2564	#16AF4h	<8000h>	32768
#33C29h	<A05h>	2565	#16B03h	<9000h>	36864
#33C83h	<A06h>	2566	#16B21h	<D000h>	53248
#64FCCh	<A11h>	2577	#16B12h	<E000h>	57344
#64FD6h	<A12h>	2578	#65094h	<70000h>	458752
#64FE0h	<A1Ah>	2586	#67D12h	<80000h>	524288
#64FEAh	<A21h>	2593	#6509Eh	<FFFFFh>	1048575

Real Numbers

#2A487h	-9.99999999999E499
#2A107h	-4.77451811461E441
#2B139h	-260
#2A42Eh	-9
#2A419h	-8
#2A404h	-7
#2A3EFh	-6
#2A3DAh	-5
#2A3C5h	-4
#2A3B0h	-3
#2A39Bh	-2
#2A386h	-1
#65002h	-0.5
#2A4B1h	-1E-499
#2A2B4h	0
#2A49Ch	1E-499
#52C72h	1E-12
#4B864h	3.49065850399E-2
#494B4h	0.1
#7D277h	4.34294481904E-1
#650BDh	0.5
#49618h	0.15
#2A2C9h	1
#31F4Fh	1.8
#2A2DEh	2
#1A223h	2.5
#650A8h	2.71828182846
#2A2F3h	3
#2A443h	3.14159265359
#2A308h	4
#2A31Dh	5
#2A332h	6
#514EBh	6.28318530718
#2A347h	7
#2A35Ch	8
#2A371h	9
#650E7h	10
#1CC03h	11
#1CC1Dh	12
#1CC37h	13
#1CC51h	14
#1CC85h	15
#1CD3Ah	16
#1CD54h	17
#1CDF2h	18
#1CE07h	19
#1CC6Bh	20
#1CCA4h	21
#1CCC3h	22
#1CCE2h	23
#1CD01h	24
#1CD20h	25
#1CD73h	26
#1CD8Dh	27
#49161h	40
#320B1h	80
#415F1h	100
#650FCh	180
#65111h	200
#2B0CEh	260
#65126h	360
#6513Bh	400
#4C035h	499
#4C068h	499
#22352h	1200
#22367h	2400
#2237Ch	4000
#0EFEEh	8192
#1A7CEh	8192
#22391h	9600
#0F003h	491520
#0F018h	29491200
#0F02Dh	707788800
#0F042h	4954521600
#2A472h	9.99999999999E499

Long Real Numbers

#2B1D6h	-1E-10000
#2B31Fh	-495.920119017593
#2B36Ch	-76.5594018140208
#2B389h	-1.21142857142857
#2A4C6h	0
#2B1BCh	1E-10000
#2A62Ch	1.74532925199433E-2
#10ED0h	7.95774715459477E-2
#2A562h	0.1
#2B3DDh	0.4
#2A57Ch	0.5
#10E68h	5.55555555555556E-1
#52A2Fh	0.7
#2B410h	9.18938533204673E-1
#2A4E0h	1
#2A4FAh	2
#5230Fh	2.30258509299405
#2A514h	3
#2A458h	3.14159265358979
#2A52Eh	4
#2A548h	5
#0F688h	6.28318530717959
#2B1FFh	7
#2B390h	9.33584905660377
#2A596h	10
#2B2DCh	12
#2B343h	30.3479606073615
#0F547h	32
#2B300h	60
#2C1C5h	100
#10E9Ch	273.15
#10EB6h	459.67
#2B0F2h	1E10000

Complex Numbers

#4AB2Ah	(0,0)
#524AFh	(0,0)
#5196Ah	(-1,0)
#524F7h	(1,0)
#5267Fh	(0,1)
#526AEh	(0,-1)

Long Complex Numbers

#5193Bh	(0,0)

Characters

#03918h	'ö'
#05127h	'A'
#05A9Eh	'A'
#0D318h	' '
#0D333h	' '
#0FA62h	'k'
#1016Fh	'ö'
#1918Ch	'i'
#44457h	'A'
#450BEh	'.'
#45174h	'-'
#6202Ch	'λ'
#6472Fh	'ö'
#6542Ch	'"'
#65433h	'#'
#6543Ah	'*'
#65441h	'+'
#65448h	','
#6544Fh	'-'
#65456h	'.'
#6545Dh	'/'
#65464h	'0'
#6546Bh	'1'
#65472h	'2'
#65479h	'3'
#65480h	'4'
#65487h	'5'
#6548Eh	'6'
#65495h	'7'
#6549Ch	'8'
#654A3h	'9'
#654AAh	':'
#654B1h	';'
#654B8h	'<'
#654BFh	'='
#654C6h	'>'
#654CDh	'A'
#654D4h	'B'
#654DBh	'C'
#654E2h	'D'
#654E9h	'E'
#654F0h	'F'
#654F7h	'G'
#654FEh	'H'
#65505h	'I'
#6550Ch	'J'
#65513h	'K'
#6551Ah	'L'
#65521h	'M'
#65528h	'N'
#6552Fh	'O'
#65536h	'P'
#6553Dh	'Q'
#65544h	'R'
#6554Bh	'S'
#65552h	'T'
#65559h	'U'
#65560h	'V'
#65567h	'W'
#6556Eh	'X'
#65575h	'Y'
#6557Ch	'Z'
#65583h	'a'
#6558Ah	'b'
#65591h	'c'
#65598h	'd'
#6559Fh	'e'
#655A6h	'f'
#655ADh	'g'
#655B4h	'h'
#655BBh	'i'
#655C2h	'j'
#655C9h	'k'
#655D0h	'l'
#655D7h	'm'
#655DEh	'n'
#655E5h	'o'
#655ECh	'p'
#655F3h	'q'
#655FAh	'r'
#65601h	's'
#65608h	't'
#6560Fh	'u'
#65616h	'v'
#6561Dh	'w'
#65624h	'x'
#6562Bh	'y'
#65632h	'z'
#65639h	'→'
#65640h	'«'
#65647h	'»'
#6564Eh	'∠'
#65655h	'∂'
#6565Ch	'∫'
#65663h	'('
#6566Ah	'←'
#65671h	'π'
#65678h	')'
#6567Fh	'Σ'
#65686h	' '
#6568Dh	' '
#65694h	'['
#6569Bh	']'
#656A2h	'{'
#656A9h	'}'
#656B0h	'≤'
#656B7h	'≥'
#656BEh	'≠'
#6947Ch	'''
#69483h	'@'
#6D2B1h	'?'
#7A929h	'α'
#7A930h	'%'
#7A937h	'''
#7A93Eh	'≤'
#7A945h	'≥'
#7A94Ch	'≠'
#7A961h	'~'
#7A968h	'β'
#7A96Fh	'▲'
#7A976h	'δ'
#7A97Dh	'ε'
#7A984h	'θ'
#7A98Bh	'γ'
#7A992h	'η'
#7A999h	'ı'
#7A9A0h	'•'
#7A9A7h	'|'
#7A9AEh	'↑'
#7A9B5h	'λ'
#7A9BCh	'μ'
#7A9C3h	'▶'
#7A9CAh	'Ω'
#7A9D1h	'←'
#7A9D8h	'↓'
#7A9DFh	'ρ'
#7A9E6h	'σ'
#7A9EDh	'τ'
#7A9F4h	'¿'
#7A9FBh	'ω'
#7AA02h	'Ξ'
#7AA09h	'±'
#7AA10h	'Π'
#7AA17h	'¢'
#7AA1Eh	'£'
#7AA25h	'¥'
#7AA2Ch	'¤'
#7AA33h	'"'
#7AA3Ah	'¦'
#7AA41h	'?'
#7AA48h	'&'
#7AA4Fh	'@'
#7AA56h	'$'

Arrays

#72000h ["Insufficient Memory" "Directory Recursion" "Undefined Local Name" "Undefined XLIB Name" "Memory Clear" "Power Lost" "Warning:" "Invalid Card Data" "Object In Use" "Port Not Available" "No Room in Port" "Object Not in Port" "Recovering Memory" "Try To Recover Memory?" "Replace RAM, Press ON" "No Mem To Config All"]

#72281h ["Bad Guess(es)" "Constant?" "Interrupted" "Zero" "Sign Reversal" "Extremum"]

#7232Ch ["Bad Packet Block Check" "Timeout" "Receive Error" "Receive Buffer Overrun" "Parity Error" "Transfer Failed" "Protocol Error" "Invalid Server Cmd." "Port Closed" "Connecting" "Retry #" "Awaiting Server Cmd." "Sending " "Receiving " "Object Discarded" "Packet #" "Processing Command" "Invalid IOPAR" "Invalid PRTPAR" "Low Battery" "Empty Stack" "Row " "Invalid Name"]

#7260Ah ["Invalid Date" "Invalid Time" "Invalid Repeat" "Nonexistent Alarm"]

#726A5h ["Invalid Unit" "Inconsistent Units"]

#72704h ["No Room to Save Stack" "Can't Edit Null Char." "Invalid User Function" "No Current Equation" "" "Invalid Syntax" "Real Number" "Complex Number" "String" "Real Array" "Complex Array" "List" "Global Name" "Local Name" "Program" "Algebraic" "Binary Integer" "Graphic" "Tagged" "Unit" "XLIB Name" "Directory" "Library" "Backup" "Function" "Command" "System Binary" "Long Real" "Long Complex" "Linked Array" "Character" "Code" "Library Data" "External" "" "LAST STACK Disabled" "LAST CMD Disabled" "HALT Not Allowed" "Array" "Wrong Argument Count" "Circular Reference" "Directory Not Allowed" "Non-Empty Directory" "Invalid Definition" "Missing Library" "Invalid PPAR" "Non-Real Result" "Unable to Isolate" "No Room to Show Stack" "Warning:←" "Error:" "Purge?" "Out of Memory" "Stack" "Last Stack" "Last Commands" "Key Assignments" "Alarms" "Last Arguments" "Name Conflict" "Command Line" ""]

#72DCFh ["Too Few Arguments" "Bad Argument Type" "Bad Argument Value" "Undefined Name" "LASTARG Disabled" "Incomplete#Subexpression" "Implicit () off" "Implicit () on"]

#72F1Eh ["Positive Underflow" "Negative Underflow" "Overflow" "Undefined Result" "Infinite Result"]

#72FE6h ["Invalid Σ Data" "Nonexistent ΣDAT" "Insufficient Σ Data" "Invalid ΣPAR" "Invalid Σ Data LN(Neg)" "Invalid Σ Data LN(0)" "Invalid EQ" "Current equation:" "No current equation." "Enter eqn, press NEW" "Name the equation,←press ENTER" "Select plot type" "Empty catalog" "undefined" "No stat data to plot" "Autoscaling" "Solving for " "No current data. Enter" "data point, press Σ+" "Select a model" "No alarms pending." "Press ALRM to create" "Next alarm:" "Past due alarm:" "Acknowledged" "Enter alarm, press SET" "Select repeat interval"

```
         "     I/O setup menu" "Plot type: " "" "" " (OFF SCREEN)"
         "Invalid PTYPE" "Name the stat data,←press ENTER"
         "Enter value (zoom out←if >1), press ENTER" "Copied to stack"
         "x axis zoom w/AUTO.←" "x axis zoom.←" "y axis zoom.←"
         "x and y axis zoom.←" "IR/wire:     " "ASCII/binary:  " "baud:      "
         "parity:          " "checksum type: " "translate code:"
         "Enter matrix, then NEW" ]

#736F9h  [ "Invalid Dimension" "Invalid Array Element" "Deleting Row"
         "Deleting Column" "Inserting Row" "Inserting Column" ]

#7AA5Dh  [ <3FFD8h> <3FFF4h> <4000Dh> <654CDh> <65583h> <7A929h> ]

#7AA94h  [ <4003Fh> <40026h> <40058h> <654D4h> <6558Ah> <7A968h> ]

#7AACBh  [ <40071h> <4008Ah> <400A3h> <654DBh> <65591h> <7A96Fh> ]

#7AB02h  [ <400BCh> <400D5h> <400EEh> <654E2h> <65598h> <7A976h> ]

#7AB39h  [ <40107h> <40120h> <40139h> <654E9h> <6559Fh> <7A97Dh> ]

#7AB70h  [ <40152h> <4016Bh> <40184h> <654F0h> <655A6h> <7A984h> ]

#7ABA7h  [ <3AD57h> <3AE33h> <1EE53h> <654F7h> <655ADh> <7A98Bh> ]

#7ABDEh  [ <3AE1Ah> <3A6E5h> <21FD1h> <654FEh> <655B4h> <7A992h> ]

#7AC15h  [ <3AB72h> <3ADA2h> <3ADBBh> <65505h> <655BBh> <7A9A0h> ]

#7AC4Ch  [ <3AF37h> <3AD70h> <3AD89h> <6550Ch> <655C2h> <7A9A7h> ]

#7AC83h  [ <3A93Dh> <3AE6Fh> <3B00Eh> <65513h> <655C9h> <7A9AEh> ]

#7ACBAh  [ <3A71Ch> <3A735h> <3B211h> <6551Ah> <655D0h> <7A9B5h> ]

#7ACF1h  [ <3A69Ah> <1A15Bh> <1A140h> <65521h> <655D7h> <7A937h> ]

#7AD28h  [ <3A992h> <20D65h> <20B40h> <65528h> <655DEh> <7A9BCh> ]

#7AD5Fh  [ <1A38Eh> <1F9C4h> <1A5E4h> <6552Fh> <655E5h> <7A9CAh> ]

#7AD96h  [ <3A834h> <1E2BAh> <3AFE6h> <65536h> <655ECh> <7A9D1h> ]

#7ADCDh  [ <3A645h> <3AE4Ch> <3AF69h> <6553Dh> <655F3h> <7A9D8h> ]

#7AE04h  [ <3A8DEh> <1FBBDh> <3A80Ch> <65544h> <655FAh> <7A9DFh> ]

#7AE3Bh  [ <1B4ACh> <1B6A4h> <1EF7Eh> <6554Bh> <65601h> <7A9E6h> ]

#7AE72h  [ <1B505h> <1B72Fh> <1F1D4h> <65552h> <65608h> <7A9EDh> ]

#7AEA9h  [ <1B55Eh> <1B79Ch> <1F2C9h> <65559h> <6560Fh> <7A930h> ]

#7AEE0h  [ <1B374h> <1B426h> <1B185h> <65560h> <65616h> <7A961h> ]

#7AF17h  [ <1B02Dh> <1BA3Dh> <1B9C6h> <65567h> <6561Dh> <7A9FBh> ]

#7AF4Eh  [ <1B278h> <1B905h> <1B94Fh> <6556Eh> <65624h> <7AA02h> ]
```

#7AF85h [<3A7F3h> <3AF50h> <3B068h> <3A7F3h> <7AA48h> <7AA4Fh>]

#7AFBCh [<3AA82h> <3A7C6h> <3B12Bh> <65575h> <6562Bh> <7AA09h>]

#7AFF3h [<3AC3Ah> <3B15Dh> <3B18Fh> <6557Ch> <65632h> <7AA10h>]

#7B02Ah [<3AB09h> <3B1DFh> <3AAEBh> <3AB09h> <7AA3Ah> <7A999h>]

#7B061h [<3A5F0h> <1FBD8h> <1FCEBh> <3A5F0h> <7AA41h> <7A9F4h>]

#7B098h [<3AA0Ah> <3B0D8h> <3B081h> <3B036h> <3E5EEh> <3E5AFh>]

#7B0CFh [<65495h> <3AE88h> <3BE54h> <65495h> <3F167h> <3F17Bh>]

#7B106h [<6549Ch> <3ADEDh> <48FD6h> <6549Ch> <3F18Fh> <3F1A3h>]

#7B13Dh [<654A3h> <3AB59h> <47B5Ah> <654A3h> <3F1B7h> <3F1CBh>]

#7B174h [<1AF05h> <3A6CCh> <3AC08h> <6545Dh> <3A6CCh> <65433h>]

#7B1ABh [<3A893h> <3A8ACh> <3A893h> <3AA37h> <3AA50h> <3AA37h>]

#7B1E2h [<65480h> <3A7A3h> <47AE7h> <65480h> <7AA56h> <7AA17h>]

#7B219h [<65487h> <3AEB5h> <3AEE2h> <65487h> <7AA1Eh> <7AA25h>]

#7B250h [<6548Eh> <3AF05h> <3AF1Eh> <6548Eh> <7AA2Ch> <7AA33h>]

#7B287h [<1ADEEh> <3ABA4h> <3A6B3h> <6543Ah> <3ABA4h> <6568Dh>]

#7B2BEh [<3A8C5h> <3A8C5h> <3A8ACh> <3AA69h> <3AA69h> <3AA50h>]

#7B2F5h [<6546Bh> <3B1CBh> <3B1B7h> <6546Bh> <7A953h> <7A94Ch>]

#7B32Ch [<65472h> <3ACF8h> <1A604h> <65472h> <654B8h> <654C6h>]

#7B363h [<65479h> <3ACBCh> <3A6FEh> <65479h> <7A93Eh> <7A945h>]

#7B39Ah [<1AD09h> <3ABD6h> <3AC21h> <6544Fh> <3AB8Dh> <6542Ch>]

#7B3D1h [<3A5CDh> <1A8BBh> <3A9CEh> <3A5CDh> <1A8BBh> <3A9CEh>]

#7B408h [<65464h> <1A8D8h> <22FEBh> <65464h> <654BFh> <65639h>]

#7B43Fh [<3A753h> <3A77Bh> <3AD04h> <3A753h> <3A77Bh> <3AD04h>]

#7B476h [<65686h> <1AABDh> <6564Eh> <65686h> <65671h> <6564Eh>]

#7B4ADh [<1AB67h> <3AB8Bh> <3ABEFh> <65441h> <3AB8Bh> <654AAh>]

#7B4E4h [<7AA5Dh> <7AA94h> <7AACBh> <7AB02h> <7AB39h> <7AB70h> <7ABA7h> <7ABDEh> <7AC15h> <7AC4Ch> <7AC83h> <7ACBAh> <7ACF1h> <7AD28h> <7AD5Fh> <7AD96h> <7ADCDh> <7AE04h> <7AE3Bh> <7AE72h> <7AEA9h> <7AEE0h> <7AF17h> <7AF4Eh> <7AF85h> <7AFBCh> <7AFF3h> <7B02Ah> <7B061h> <7B098h> <7B0CFh> <7B106h> <7B13Dh> <7B174h> <7B1ABh> <7B1E2h> <7B219h> <7B250h> <7B287h> <7B2BEh> <7B2F5h> <7B32Ch> <7B363h> <7B39Ah> <7B3D1h> <7B408h> <7B43Fh> <7B476h> <7B4ADh>]

Strings

```
#055DFh   ""
#0E524h   ""
#0E5C6h   ""
#0FA69h   "g"
#0FA8Eh   "m"
#0FAAEh   "A"
#0FACEh   "s"
#0FAEEh   "K"
#0FB0Eh   "cd"
#0FB30h   "mol"
#0FB54h   "?"
#11231h   "1E"
#15331h   "NXEQ"
#15442h   ":  "
#1585Fh   "EXPR="
#158B9h   "LEFT"
#158E4h   "RIGHT"
#15911h   "EXPR"
#15DF6h   "  x  "
#15E47h   "GROB  "
#15F23h   "C#  "
#15FB5h   "C#  "
#161B2h   "XLIB  "
#16BB2h   "{"
#16BEBh   "}"
#16C42h   "DIR"
#16CEDh   "END"
#16D25h   ":"
#17DB4h   "PICT"
#183C9h   "0: "
#19AABh   ""
#19F4Fh   "Rpt="
#19F70h   "  week(s)"
#19F8Ah   "  day(s)"
#19FA2h   "  hour(s)"
#19FBCh   "  minute(s)"
#19FDAh   "  second(s)"
#19FF8h   "  ticks"
#1FF34h   "Intercept"
#1FF50h   "Slope"
#20E4Bh   "  "
#2127Dh   "IO"
#2189Dh   "ROM"
#218C6h   "SYSRAM"
#2212Dh   "IR"
#2213Bh   "wire"
#2216Bh   "binary"
#22181h   "ASCII"
#221F9h   "none  "
#2220Dh   "odd  "
#2221Fh   "even  "
#22233h   "mark  "
#2226Ah   "spc  "
#22290h   "invalid"
#22ED7h   "IF-prompt"
#2539Ah   "TO"
#253A8h   "DIR"
#253B8h   ":"
#253C4h   "ELSE"
#253D6h   "END"
#253E6h   "UNTIL"
#253FAh   "REPEAT"
#25410h   "NEXT"
#25422h   "STEP"
#25434h   "THEN"
#25446h   "→"
#25678h   "bodh"
#25CF5h   "    "
#28A08h   "SQRT"
#28A1Ah   "SQ"
#28A28h   "INV"
#2970Ah   "Invalid Expression"
#2DF88h   ":←"
#2E4F0h   "←←"
#2E878h   "F"
#2E887h   "G"
#2E8B1h   "R"
#2F162h   "%%HP: "
#31D26h   "       "
#32341h   "  "
#34105h   "+   "
#34115h   "-   "
#3412Fh   "?   "
#3971Dh   "HALT"
#397D9h   "1USR"
#397EBh   "USER"
#3982Ah   "ALG"
#39862h   "PRG"
#3998Ah   "  }"
#39F09h   "1:  "
#39F28h   "     "
#39FF2h   "_"
#3B29Dh   "PARTS"
#3B2C0h   "PROB"
#3B2E1h   "HYP"
#3B300h   "MATRX"
#3B323h   "VECTR"
#3B346h   "BASE"
#3B55Bh   "STK"
#3B57Ah   "OBJ"
#3B599h   "DSPL"
#3B5BAh   "CTRL"
#3B5DBh   "BRCH"
#3B5FCh   "TEST"
#3B65Eh   "DRPN"
#3B7ECh   "DBUG"
#3B80Dh   "SST"
#3B82Ch   "SST↓"
#3B84Dh   "NEXT"
#3BA12h   "IR/W"
#3BA33h   "ASCII"
#3BB6Eh   "SYM"
#3BBB5h   "BEEP"
```

#3BC0Dh	"CNCT"
#3BE38h	"SOLVR"
#3BED1h	"PLOTR"
#3BF30h	"PTYPE"
#3BF76h	"NEW"
#3BFA4h	"EDEQ"
#3C001h	"CAT"
#3C066h	"HIST"
#3C366h	"RESET"
#3C4E2h	"SET"
#3C529h	"ADJST"
#3C574h	"ALRM"
#3C67Bh	"HR+"
#3C6A9h	"HR-"
#3C6D7h	"MIN+"
#3C707h	"MIN-"
#3C737h	"SEC+"
#3C767h	"SEC-"
#3C7ABh	">DATE"
#3C7CEh	">TIME"
#3C7F1h	"A/PM"
#3C812h	"EXEC"
#3C851h	"RPT"
#3C88Eh	"SET"
#3C8DFh	"WEEK"
#3C900h	"DAY"
#3C91Fh	"HOUR"
#3C940h	"MIN"
#3C95Fh	"SEC"
#3C97Eh	"NONE"
#3C9C2h	"→DATE"
#3C9E5h	"→TIME"
#3CA08h	"A/PM"
#3CA3Dh	"12/24"
#3CA74h	"M/D"
#3CB4Ch	"NEW"
#3CB7Ah	"EDITΣ"
#3CDA5h	"LIN"
#3CDC4h	"LOG"
#3CDE3h	"EXP"
#3CE02h	"PWR"
#3CE21h	"BEST"
#3CE7Eh	"LENG"
#3CE9Fh	"AREA"
#3CEC0h	"VOL"
#3CEDFh	"TIME"
#3CF00h	"SPEED"
#3CF23h	"MASS"
#3CF44h	"FORCE"
#3CF67h	"ENRG"
#3CF88h	"POWR"
#3CFA9h	"PRESS"
#3CFCCh	"TEMP"
#3CFEDh	"ELEC"
#3D00Eh	"ANGL"
#3D02Fh	"LIGHT"
#3D066h	"VISC"
#3D09Bh	"m"
#3D0A7h	"cm"
#3D0B5h	"mm"
#3D0C3h	"yd"
#3D0D1h	"ft"
#3D0DFh	"in"
#3D0EDh	"Mpc"
#3D0FDh	"pc"
#3D10Bh	"lyr"
#3D11Bh	"au"
#3D129h	"km"
#3D137h	"mi"
#3D145h	"nmi"
#3D155h	"miUS"
#3D167h	"chain"
#3D17Bh	"rd"
#3D189h	"fath"
#3D19Bh	"ftUS"
#3D1ADh	"mil"
#3D1BDh	"µ"
#3D1C9h	"Å"
#3D1D5h	"fermi"
#3D202h	"m^2"
#3D212h	"cm^2"
#3D224h	"b"
#3D230h	"yd^2"
#3D242h	"ft^2"
#3D254h	"in^2"
#3D266h	"km^2"
#3D278h	"ha"
#3D286h	"a"
#3D292h	"mi^2"
#3D2A4h	"miUS^2"
#3D2BAh	"acre"
#3D2E5h	"m^3"
#3D2F5h	"st"
#3D303h	"cm^3"
#3D315h	"yd^3"
#3D327h	"ft^3"
#3D339h	"in^3"
#3D34Bh	"l"
#3D357h	"galUK"
#3D36Bh	"galC"
#3D37Dh	"gal"
#3D38Dh	"qt"
#3D39Bh	"pt"
#3D3A9h	"ml"
#3D3B7h	"cu"
#3D3C5h	"ozfl"
#3D3D7h	"ozUK"
#3D3E9h	"tbsp"
#3D3FBh	"tsp"
#3D40Bh	"bbl"
#3D41Bh	"bu"
#3D429h	"pk"
#3D437h	"fbm"
#3D460h	"yr"
#3D46Eh	"d"
#3D47Ah	"h"

```
#3D486h   "min"
#3D496h   "s"
#3D4A2h   "Hz"
#3D4C9h   "m/s"
#3D4D9h   "cm/s"
#3D4EBh   "ft/s"
#3D4FDh   "kph"
#3D50Dh   "mph"
#3D51Dh   "knot"
#3D52Fh   "c"
#3D53Bh   "ga"
#3D562h   "kg"
#3D570h   "g"
#3D57Ch   "lb"
#3D58Ah   "oz"
#3D59Bh   "slug"
#3D5AAh   "lbt"
#3D5BAh   "ton"
#3D5CAh   "tonUK"
#3D5DEh   "t"
#3D5EAh   "ozt"
#3D5FAh   "ct"
#3D608h   "grain"
#3D61Ch   "u"
#3D628h   "mol"
#3D651h   "N"
#3D65Dh   "dyn"
#3D66Dh   "gf"
#3D67Bh   "kip"
#3D688h   "lbf"
#3D69Bh   "pdl"
#3D6C4h   "J"
#3D6D0h   "erg"
#3D6E0h   "Kcal"
#3D6F2h   "cal"
#3D702h   "Btu"
#3D712h   "ft*lbf"
#3D728h   "therm"
#3D73Ch   "MeV"
#3D74Ch   "eV"
#3D773h   "W"
#3D77Fh   "hp"
#3D7A6h   "Pa"
#3D7B4h   "atm"
#3D7C4h   "bar"
#3D7D4h   "psi"
#3D7E4h   "torr"
#3D7F6h   "mmHg"
#3D808h   "inHg"
#3D81Ah   "inH2O"
#3D847h   "°C"
#3D855h   "°F"
#3D863h   "K"
#3D86Fh   "°R"
#3D896h   "V"
#3D8A2h   "A"
#3D8AEh   "C"
#3D8BAh   "Ω"
#3D8C6h   "F"
#3D8D2h   "W"
#3D8DEh   "Fdy"
#3D8EEh   "H"
#3D8FAh   "mho"
#3D90Ah   "S"
#3D916h   "T"
#3D922h   "Wb"
#3D949h   "°"
#3D955h   "r"
#3D961h   "grad"
#3D973h   "arcmin"
#3D989h   "arcs"
#3D99Bh   "sr"
#3D9C2h   "fc"
#3D9D0h   "flam"
#3D9E2h   "lx"
#3D9F0h   "ph"
#3D9FEh   "sb"
#3DA0Ch   "lm"
#3DA1Ah   "cd"
#3DA28h   "lam"
#3DA51h   "Gy"
#3DA5Fh   "rad"
#3DA6Fh   "rem"
#3DA7Fh   "Sv"
#3DA8Dh   "Bq"
#3DA9Bh   "Ci"
#3DAA9h   "R"
#3DACEh   "P"
#3DADAh   "St"
#3DB6Ah   "HEX"
#3DBACh   "DEC"
#3DBEEh   "OCT"
#3DC30h   "BIN"
#3DECAh   "FCN"
#3DF43h   "ROOT"
#3DF64h   "ISECT"
#3DF87h   "SLOPE"
#3DFAAh   "AREA"
#3DFCBh   "EXTR"
#3E01Eh   "F(X)"
#3E03Fh   "F'"
#3E05Ch   "NXEQ"
#3E0A0h   "+/-"
#3E0C9h   "REPL"
#3E0EAh   "SUB"
#3E109h   "DEL"
#3E128h   "COORD"
#3E17Dh   "ZOOM"
#3E1A3h   "Keys"
#3E1C4h   "MARK"
#3E221h   "CIRCLE"
#3E246h   "Z-BOX"
#3E282h   "DOT+"
#3E2B7h   "DOT-"
#3E2E2h   "←SKIP"
#3E364h   "SKIP→"
```

#3E3E6h	"←DEL"
#3E4CFh	"DEL→"
#3E595h	"INS"
#3E5D2h	"↑STK"
#3E729h	"SETUP"
#3E783h	"STD"
#3E7C5h	"FIX"
#3E7F8h	"SCI"
#3E828h	"ENG"
#3E85Eh	"ML"
#3E89Eh	"DEG"
#3EA5Ch	"CMD"
#3EAADh	"STK"
#3EB28h	"ARG"
#3EB77h	"FM, "
#3EB8Eh	"CLK"
#3EC05h	"MODL"
#3F40Ch	"PORT0"
#3F461h	"PORT1"
#3F4B8h	"PORT2"
#414BDh	": "
#415A7h	": "
#4302Eh	"VIEW"
#430C7h	"DRPN"
#430E8h	"KEEP"
#43E09h	"LEVEL"
#44228h	"▶"
#44243h	":"
#46017h	"EDIT"
#4604Ch	"←WID"
#4606Dh	"WID→"
#460B6h	"+ROW"
#460D7h	"-ROW"
#460F8h	"+COL"
#46119h	"-COL"
#4613Ah	"→STK"
#4615Bh	"↑STK"
#461C2h	"GO→"
#461F0h	"GO↓"
#4621Eh	"VEC"
#47660h	"Indep:"
#476BCh	"Depnd:"
#476F0h	"x:"
#4771Ch	"y:"
#47900h	".EQ"
#47910h	",EQ"
#47F6Fh	"["
#47F8Fh	"x"
#47FAFh	"]"
#47FD4h	" dir"
#48009h	":"
#48042h	"▶"
#485C1h	"1-VAR"
#48657h	"PLOT"
#486B9h	"2-VAR"
#4871Dh	"EDIT"
#48766h	"SOLVR"
#487CAh	"PLOTR"
#48810h	"EQ+"
#4891Fh	"EDIT"
#48995h	"PURGE"
#48A76h	"→STK"
#48ACEh	"undefined"
#48B44h	"VIEW"
#48C69h	"FAST"
#48CBCh	"ORDER"
#48D25h	"EDIT"
#48D92h	"PURGE"
#48DECh	"EXECS"
#4971Dh	": "
#49825h	"Slope"
#49870h	"Root"
#4991Dh	"Area"
#499FCh	"I-sect"
#49A4Eh	"Extrm"
#49A8Fh	"F(x)"
#4A605h	")="
#4A677h	"Xcol:"
#4A69Ah	" Ycol:"
#4A6C4h	" Modl:"
#4EE51h	"XAUTO"
#4EE92h	"X"
#4EEC6h	"Y"
#4EEFAh	"XY"
#596B5h	"COLCT"
#59701h	"DNEG"
#5974Fh	"DINV"
#5979Dh	"*1"
#59823h	"^1"
#5986Dh	"/1"
#598F3h	"+1-1"
#59955h	"←→"
#59981h	"←A"
#599D0h	"A→"
#59A1Fh	"←T"
#59A6Eh	"T→"
#59ABDh	"(←"
#59B0Ch	"→)"
#59B5Bh	"(())"
#59B88h	"AF"
#59BB7h	"←M"
#59C06h	"M→"
#59C55h	"-()"
#59C83h	"1/()"
#59CB3h	"E()"
#59CE1h	"L()"
#59D0Fh	"L*"
#59D38h	"E^"
#59D67h	"→()"
#59D88h	"←D"
#59E07h	"D→"
#59E56h	"→TRG"
#59E86h	"→()"
#59ED7h	"→DEF"
#59F07h	"TRG*"
#65150h	"]"

```
#6515Ch   "[  "
#6516Ah   "["
#65176h   "{"
#65182h   "}"
#6518Eh   "#"
#6519Ah   "  "
#651A6h   "$"
#651B2h   "&"
#651BEh   "  "
#651CAh   "»"
#651D6h   "«"
#651E2h   "E"
#651EEh   "∠"
#651FAh   "Σ"
#6520Eh   "|"
#65212h   "                    "
#65238h   "←"
#65244h   "der"
#65254h   "  "
#65260h   "UNKNOWN"
#65278h   """
#65284h   "'"
#65290h   ", "
#6529Ch   ". "
#652A8h   "; "
#652B4h   "("
#652C0h   ")"
#652CCh   "^"
#652D8h   "*"
#652E4h   "/"
#652F0h   "+"
#652FCh   "-"
#65308h   "="
#65314h   "√"
#65320h   "∂"
#6532Ch   "GROB"
#6533Eh   "C$"
#6534Ch   "0"
#65358h   "1"
#65364h   "2"
#65370h   "3"
#6537Ch   "4"
#65388h   "5"
#65394h   "6"
#653A0h   "7"
#653ACh   "8"
#653B8h   "9"
#656C5h   "R∠∠"
#656D5h   "R∠Z"
#656E5h   "XYZ"
#656F5h   "«»"
#65703h   "{}"
#65711h   "[]"
#6571Fh   "''"
#6572Dh   "::"
#6573Bh   "()"
#65749h   """"
#65757h   "ECHO"
#65769h   "EXIT"
#6577Bh   "Undefined"
#65797h   "RAD"
#657A7h   "GRAD"
#67365h   "d"
#69692h   " RATIO "
#6A577h   "RULES"
#6A59Fh   "EDIT"
#6A5C0h   "EXPR"
#6A5E1h   "SUB"
#6A600h   "REPL"
#6B413h   "NOT "
```

Binary Integers

```
#10E34h   #0000000000000000h
#10E4Eh   #0000000100000000h
#1A215h   #0000h
#1A471h   #0526260410h
#1A9F9h   #8010h
#1AC75h   #70107h
#1AEDEh   #80108h
#1B013h   #0504108h
#1B104h   #90109h
#1B113h   #0504109h
#1B3E7h   #C010h
#1BB2Ah   #010Ah
#1BE84h   #80108h
#1E7CEh   #50105h
#1E854h   #40104h
#1E8CBh   #5010h
#1EA21h   #60106h
#1F00Eh   #0524310h
#1F241h   #053626060410h
#1F319h   #052606073410h
#1F40Ch   #05B0734122h
#1F523h   #052410h
#1F5D9h   #059063410h
#1FB1Dh   #72109h
#26DD2h   #0590410h
#26DE3h   #050410h
#26DF3h   #05060410h
#26E05h   #0506060410h
#26E19h   #050606060410h
#26E2Fh   #05060606060410h
#26E47h   #00700h
```

Lists

```
#055E9h   { }
#085A8h   { }
#0E475h   { M N }
#0FA53h   { '1_kg' '1_m' '1_A'
          '1_s' '1_K' '1_cd' '1_mol'
          '1_?' }
#101D5h   { }
#1085Eh   { }
#10868h   { }
#10872h   { }
#1087Ch   { }
#10886h   { }
#14396h   { 'halt }
#155EFh   { 'nohalt }
#19A91h   { 0 "" 0 }
#19EFAh   { 4954521600 707788800
          29491200 491520 8192 1 }
#19F68h   { " week(s)" " day(s)" "
          hour(s)" " minute(s)"
          " second(s)" " ticks" }
#1F960h   { 'num }
#1FF2Fh   { "Intercept" "Slope" }
#221F4h   { "none " "odd " "even "
          "mark " }
#2234Dh   { 1200 2400 4800 9600 }
#223C9h   { 1 2 3 }
#22400h   { 0 1 2 3 4 }
#22441h   { 0 1 2 3 }
#23754h   { 'noname 'stop }
#23879h   { 'ioinprogress }
#23903h   { st ofs tok }
#23989h   { }
#24A28h   { i j }
#25699h   { <2h> <8h> <Ah> <10h> }
#258EEh   { }
#25A06h   { '1 '2 '3 }
#272C8h   { 'ttt 'str 'ofs 'tok 'rbu
          'idfflg 'tmpop 'tmppdat
          'ploc 'bu 'unbound }
#2807Ah   { }
#281D3h   { i * + }
#28884h   { { } }
#2C756h   { 1 2 0 0 LINFIT }
#2D408h   { }
#2D5F5h   { 'PACKET 'RETRY 'MaxR }
#2D839h   { 'KP 'PKNO }
#2D868h   { 'LNAME 'RETRY 'KMODE
          'KRM
#2DD56h   { 'ERRMSG }
#2DF01h   { 'LNAME 'KMODE 'KRM }
#2E6CDh   { 'KP 'PKNO }
#2E8E5h   { 'KP 'PKNO }
#2E90Dh   { 'LNAME 'KMODE 'KRM }
#2E99Eh   { 9600 0 0 0 3 1 }
#2E9F0h   { }
#2EED3h   { 'LNAME 'OBJ 'PACKET
          'KRM }
#2F1FDh   { 'KLIST 'OPOS 'KML }
#2F6F0h   { 'RETRY }
#31C32h   { 'IWrap }
#31CA9h   { 1 }
#31F4Ah   { 1.8 "" 80 "←" }
#32F9Fh   { 'nohalt }
#3304Eh   { <Eh> }
#3306Ch   { }
#34D28h   { }
#36796h   { <3h> }
#368CCh   { 3 }
#36BF1h   { #a #b }
#36CEAh   { #b }
#36D5Ah   { }
#36D82h   { }
#38A39h   { 'SavedUI }
#3B290h   { "PARTS" { ABS SIGN CONJ
          ARG RE IM MIN MAX MOD %
          %CH %T MANT XPON IP FP
          FLOOR CEIL RND TRNC MAXR
          MINR } }
#3B2DCh   { "HYP" { SINH ASINH COSH
          ACOSH TANH ATANH EXPM
          LNP1 } }
#3B2FBh   { "MATRX" { CON IDN TRN
          RDM DET RSD ABS RNRM CNRM
          } }
#3B36Ch   { ABS SIGN CONJ ARG RE IM
          MIN MAX MOD % %CH %T MANT
          XPON IP FP FLOOR CEIL RND
          TRNC MAXR MINR }
#3B420h   { SINH ASINH COSH ACOSH
          TANH ATANH EXPM LNP1 }
#3B452h   { CON IDN TRN RDM DET RSD
          ABS RNRM CNRM }
#3B556h   { "STK" { OVER ROT ROLL
          ROLLD PICK DEPTH DUP DUP2
          DUPN DROP2 { "DRPN" DROPN
          } } }
#3B575h   { "OBJ" { OBJ→ EQ→ →ARRY
          →LIST →STR →TAG R→C C→R
          DTAG →UNIT TYPE VTYPE SIZE
          POS REPL SUB NUM CHR
          PUT GET PUTI GETI } }
```

```
#3B5F7h  { "TEST" { AND OR XOR NOT
         SAME TYPE == ≠ < > ≤ ≥ SF
         CF FS? FC? FS?C FC?C } }
#3B622h  { OVER ROT ROLL ROLLD PICK
         DEPTH DUP DUP2 DUPN
         DROP2 { "DRPN" DROPN } }
#3B659h  { "DRPN" DROPN }
#3B67Fh  { OBJ→ EQ→ →ARRY →LIST
         →STR →TAG R→C C→R DTAG
         →UNIT TYPE VTYPE SIZE POS
         REPL SUB NUM CHR PUT GET
         PUTI GETI }
#3B90Eh  { AND OR XOR NOT SAME TYPE
         == ≠ < > ≤ ≥ SF CF FS?
         FC? FS?C FC?C }
#3B972h  { PR1 PRST PRSTC PRLCD
         PRVAR CR DELAY OLDPRT }
#3BC8Dh  { ASN STOKEYS RCLKEYS
         DELKEYS MENU CST TMENU
         RCLMENU STOF RCLF SF CF
         FS? FC? FS?C FC?C }
#3BCE7h  { MEM BYTES VARS ORDER
         PATH CRDIR TVARS PVARS
         NEWOB LIBS ATTACH DETACH
         MERGE FREE ARCHIVE
         RESTORE PGDIR }
#3BD46h  { STO+ STO- STO* STO/ INCR
         DECR SINV SNEG SCONJ }
#3C03Eh  { FUNCTION CONIC POLAR
         PARAMETRIC TRUTH BAR {
         "HIST" HISTOGRAM }
         SCATTER }
#3C061h  { "HIST" HISTOGRAM }
#3C483h  { COLCT EXPAN ISOL QUAD
         SHOW TAYLR ↑MATCH ↓MATCH
         | APPLY QUOTE →Qπ }
#3C9BDh  { "→DATE" →DATE }
#3C9E0h  { "→TIME" →TIME }
#3CD9Bh  { { "LIN" LINFIT } { "LOG"
         LOGFIT } { "EXP" EXPFIT
         } { "PWR" PWRFIT } {
         "BEST" BESTFIT } }
#3CDA0h  { "LIN" LINFIT }
#3CDBFh  { "LOG" LOGFIT }
#3CDDEh  { "EXP" EXPFIT }
#3CDFDh  { "PWR" PWRFIT }
#3CE1Ch  { "BEST" BESTFIT }
#3D096h  { "m" "cm" "mm" "yd" "ft"
         "in" "Mpc" "pc" "lyr"
         "au" "km" "mi" "nmi" "miUS"
         "chain" "rd" "fath" "ftUS"
         "mil" "μ" "Å" "fermi" }
```

```
#3D1FDh  { "m^2" "cm^2" "b" "yd^2"
         "ft^2" "in^2" "km^2" "ha"
         "a" "mi^2" "miUS^2"
         "acre" }
#3D2E0h  { "m^3" "st" "cm^3" "yd^3"
         "ft^3" "in^3" "l"
         "galUK" "galC" "gal" "qt"
         "pt" "ml" "cu" "ozfl"
         "ozUK" "tbsp" "tsp" "bbl"
         "bu" "pk" "fbm" }
#3D45Bh  { "yr" "d" "h" "min" "s"
         "Hz" }
#3D4C4h  { "m/s" "cm/s" "ft/s"
         "kph" "mph" "knot" "c"
         "ga" }
#3D55Dh  { "kg" "g" "lb" "oz"
         "slug" "lbt" "ton" "tonUK"
         "t" "ozt" "ct" "grain" "u"
         "mol" }
#3D64Ch  { "N" "dyn" "gf" "kip"
         "lbf" "pdl" }
#3D6BFh  { "J" "erg" "Kcal" "cal"
         "Btu" "ft*lbf" "therm"
         "MeV" "eV" }
#3D76Eh  { "W" "hp" }
#3D7A1h  { "Pa" "atm" "bar"
         "psi" "torr" "mmHg"
         "inHg" "inH2O" }
#3D842h  { "°C" "°F" "K" "°R" }
#3D891h  { "V" "A" "C" "Ω" "F" "W"
         "Fdy" "H" "mho" "S" "T"
         "Wb" }
#3D944h  { "°" "r" "grad" "arcmin"
         "arcs" "sr" }
#3D9BDh  { "fc" "flam" "lx" "ph"
         "sb" "lm" "cd" "lam" }
#3DA4Ch  { "Gy" "rad" "rem" "Sv"
         "Bq" "Ci" "R" }
#3DAC9h  { "P" "St" }
#3DAF2h  { CONVERT UBASE UVAL UFACT
         →UNIT }
#4132Dh  { "0" "1" "2" "&" }
#4330Bh  { "" <0h> <1h> <1h> <2h>
         <0h> }
#45716h  { <1h> <1h> }
#47B73h  { 9 15 }
#47BBEh  { 3 15 }
#49199h  { }
#4A7CFh  { LINFIT LOGFIT EXPFIT
         PWRFIT }
#4C93Fh  { 'xmax 'N }
#4CF50h  { 'EnvOK 'EXITFCN }
```

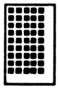
```
#4FF90h  { 'xe 'ye 'x 'y 'xc 'yc
         'r2 'left 'up 'exit }
#50039h  { 'PlotEnv }
#52D26h  { }
#5456Ah  { 'tcls 'fcls }
#547ABh  { }
#549CCh  { 'dvar }
#54DCBh  { 'xSYMfcn 'xfcn }
#55666h  { 'oth }
#5577Eh  { 'scl 'xSYMfcn 'xfcn }
#557FBh  { 'xSYMfcn 'xfcn }
#56971h  { 'sumexpr 'sumvar }
#56BCDh  { 2 ^ / }
#56E43h  { π 180 / * }
#56E61h  { π 200 / * }
#56EA2h  { 180 π / * }
#56EC0h  { 200 π / * }
#56EFCh  { 'dv }
#571F2h  { 'dv 'op 'nm }
#576F8h  { π i * }
#5772Fh  { 2 π * i * }
#578CEh  { 'ni 'ns }
#57EA3h  { '*s }
#580F9h  { '+s }
#58DB1h  { 'fl }
#59110h  { 'nmls }
#592B8h  { 'c 'b 'a }
#59512h  { 'n 'prog }
#59641h  { 'n }
#59919h  { 1 + 1 - }
#5A60Fh  { 'piflag }
#5A660h  { 'd 'r }
#5A75Ch  { 'd 'R 'est 'X 'T }
#5AAE0h  { 'bnds 'dvar }
#5D678h  { 'which 'op1 'op2 }
#5DD42h  { 2 ^ - √ i }
#5DE1Eh  { π / * }
#5FDBCh  { 'ct 'pp 'ep }
#600F3h  { }
#60806h  { 'reg 'sur 'cts 'sun 'mlg
         'ckd 'prd 'prp 'rhs }
#60BE4h  { 'patternls 'compos
         'varls }
#60E73h  { { } { } { } { } &1 &2
         &3 &4 }
#6419Ah  { <13Eh> <123h> <DFFh> }
#68484h  { }
#69A92h  { 'Radix 'KeysOK? 'ExprLit
         'BuffW 'BuffH 'SaveBlank
         'ManOp 'nohalt 'AppMode
         'NameGrob 'EXITFCN
         'FontGauge 'LE 'LB 'TE
         'FormEnvOK 'prow 'pcol
         'cursy 'cursx 'ttt 'source
         'ofs 'tok 'rbv 'idfflg
         'tmpop 'tmppdat 'ploc 'bv
         'unbound }
```

Units

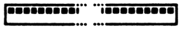
```
#0FA58h  '1_kg'
#0FA84h  '1_m'
#0FAA4h  '1_A'
#0FAC4h  '1_s'
#0FAE4h  '1_K'
#0FB04h  '1_cd'
#0FB26h  '1_mol'
#0FB4Ah  '1_?'
```

Graphics Objects

#13DB4h GROB 6 10 F1F1F1F1F1F1F1F1F100

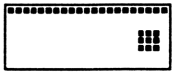


#39B20h GROB 131 2 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0000000000000000000000000000000000

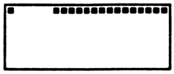


#3A337h GROB 21 8 FFFFFF0000000000000000000000000000
00000000000000

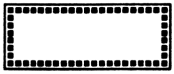
#3A399h GROB 21 8 FFFFFF0000000000000000E00000E00000
E0000000000000



#3A3FBh GROB 21 8 1CFFFF0000000000000000000000000000
00000000000000



#3A45Dh GROB 21 8 FFFFFF1000011000011000011000011000
01100001FFFFFF

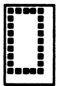


#5053Ch GROB 5 5 4040F14040

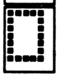
#5055Ah GROB 5 5 11A040A011

#505B2h GROB 0 0

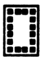
#66EA5h GROB 6 10 F111111111111111F100



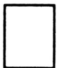
#66ECDh GROB 6 8 F11111111111F100



#66EF1h GROB 4 6 F090909090F0

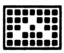


#66F11h GROB 6 8 0000000000000000



#66F35h GROB 6 10 00000000000000000000

#66F5Dh GROB 7 5 F777B6D5F7F7



#66F7Dh GROB 5 4 F1B151F1

Global Names

```
#0DF01h  'Alarms'
#0DF28h  'Alarms'
#1576Ch  'EQ'
#15781h  ''
#19A72h  'ALRMDAT'
#19B1Fh  'ALRMDAT'
#19D8Eh  'ALRMDAT'
#211B4h  'CST'
#225A4h  'S'
#2C1FDh  'ΣDAT'
#2C738h  'ΣPAR'
#2E9D5h  'IOPAR'
#2EA59h  'IOPAR'
#31F87h  'PRTPAR'
#31FB8h  'PRTPAR'
#34D88h  ''symb'
#3FACFh  'SKEY'
#40938h  'αENTER'
#409DFh  'βENTER'
#41A43h  'UserKeys'
#41A69h  'UserKeys.CRC'
#41B07h  'S'
#4353Eh  'ALG'
#4358Ah  'α'
#435CEh  'V'
#47459h  'X'
#4804Bh  'ALRMDAT'
#4A145h  'X'
#4A19Eh  'X'
#4A1DEh  'X'
#4A22Dh  'X'
#4A25Eh  'X'
#4AB1Ch  'X'
#4AB59h  'Y'
#50FCEh  'X'
#50FE6h  'Y'
#51288h  'PPAR'
#51436h  's1'
#56859h  'IERR'
#5793Fh  'n0'
#5795Dh  's0'
#59304h  's1'
```

Local Names

```
#0E47Ah  'M'
#0E483h  'N'
#0E4A0h  'M'
#0E4AEh  'N'
#0E4C1h  'M'
#1439Bh  ''halt'
#14483h  ''nohalt'
#1F96Fh  ''num'
#1F97Eh  ''fcn'
#2372Eh  ''stop'
#2373Fh  ''noname'
#2387Eh  ''ioinprogress'
#23908h  'st'
#23913h  'ofs'
#23920h  'tok'
#2394Bh  'st'
#23956h  'ofs'
#23963h  'tok'
#24A2Dh  'i'
#24A36h  'j'
#24A5Dh  'i'
#24A6Bh  'j'
#24B0Ah  'j'
#24B1Dh  'i'
#24B30h  'i'
#24BB6h  'j'
#24BD3h  'i'
#24BE1h  'i'
#25A08h  ''1'
#25A16h  ''2'
#25A21h  ''3'
#25A3Bh  ''1'
#25A46h  ''2'
#25A51h  ''3'
#272CDh  ''ttt'
#272DCh  ''str'
#272EBh  ''ofs'
#272FAh  ''tok'
#27309h  ''rbv'
#27318h  ''idfflg'
#2732Dh  ''tmpop'
#27340h  ''tmppdat'
#27357h  ''ploc'
#27368h  ''bv'
#27375h  ''unbound'
#203A0h  ''PKNO'
#203B1h  ''PACKET'
#203C6h  ''RETRY'
#203D9h  ''ERRMSG'
#203EEh  ''KP'
#203FBh  ''LNAME'
#2040Eh  ''OBJ'
#2041Dh  ''OPOS'
#2042Eh  ''EXCHP'
#2045Ah  ''KLIST'
#2046Dh  ''KMODE'
#20480h  ''KPTRN'
#20493h  ''KRM'
#204A2h  ''MaxR'
#2F211h  ''KML'
#2F46Eh  ''KEOF'
#31C37h  ''IWrap'
#34D30h  ''
#36BF6h  '#a'
#36C01h  '#b'
#36C2Fh  '#b'
#36C3Fh  '#a'
#36CEFh  '#b'
#36D18h  '#b'
#38A3Eh  ''SavedUI'
#3FAE8h  'SKEY'
```

```
#41BEAh   'S'
#43555h   'ALG'
#4359Dh   'α'
#435E1h   'V'
#4C944h   ''xmax'
#4C955h   ''N'
#4CF55h   ''EnvOK'
#4CF68h   ''EXITFCN'
#4D30Dh   ''EnvOK'
#4D352h   ''EnvOK'
#4D36Fh   ''EnvOK'
#4FF9Dh   ''xe'
#4FFAAh   ''ye'
#4FFB7h   ''x'
#4FFC2h   ''y'
#4FFCDh   ''xc'
#4FFDAh   ''yc'
#4FFE7h   ''r2'
#4FFF4h   ''left'
#50005h   ''up'
#50012h   ''exit'
#5003Eh   ''PlotEnv'
#5190Bh   ''PlotEnv'
#5456Fh   ''tcls'
#54580h   ''fcls'
#5460Eh   ''tcls'
#54624h   ''fcls'
#5465Dh   ''tcls'
#5466Eh   ''fcls'
#549DBh   ''dvar'
#54DD0h   ''xSYMfcn'
#54DE7h   ''xfcn'
#5566Bh   ''oth'
#55783h   ''scl'
#55792h   ''xSYMfcn'
#557A9h   ''xfcn'
#55800h   ''xSYMfcn'
#55817h   ''xfcn'
#56976h   ''sumexpr'
#5698Dh   ''sumvar'
#56F0Bh   ''dv'
#5720Bh   ''nm'
#57218h   ''op'
#578E2h   ''ni'
#578EFh   ''ns'
#57EF3h   ''*s'
#58149h   ''+s'
#59115h   ''nmls'
#592C0h   ''c'
#592CBh   ''b'
#592D6h   ''a'
#59517h   ''n'
#59522h   ''prog'
#59646h   ''n'
#5A614h   ''piflag'
#5A665h   ''d'
#5A670h   ''r'
#5A761h   ''d'
#5A76Ch   ''R'
#5A777h   ''est'
#5A786h   ''X'
#5A791h   ''T'
#5AAE5h   ''bnds'
#5D67Dh   ''which'
#5D690h   ''op1'
#5D69Fh   ''op2'
#5FDC1h   ''ct'
#5FDCEh   ''pp'
#5FDDBh   ''ep'
#6080Bh   ''reg'
#6081Ah   ''sur'
#60829h   ''cts'
#60838h   ''sun'
#60847h   ''mlg'
#60856h   ''ckd'
#60865h   ''prd'
#60874h   ''prp'
#60883h   ''rhs'
#60BE9h   ''patternls'
#60C04h   ''compos'
#60C19h   ''varls'
#60C4Fh   ''patternls'
#60C6Ah   ''compos'
#60CCAh   ''compos'
#60D7Fh   ''varls'
#60E8Ch   '&1'
#60E97h   '&2'
#60EA2h   '&3'
#60EADh   '&4'
#61D3Ah   ''
#69A97h   ''Radix'
#69AAAh   ''KeysOK?'
#69AC1h   ''ExprLit'
#69AD8h   ''BuffW'
#69AEBh   ''BuffH'
#69AFEh   ''SaveBlank'
#69B19h   ''ManOp'
#69B2Ch   ''nohalt'
#69B41h   ''AppMode'
#69B58h   ''NameGrob'
#69B71h   ''EXITFCN'
#69B88h   ''FontGauge'
#69BA3h   ''LE'
#69BB0h   ''LB'
#69BBDh   ''TE'
#69BCAh   ''FormEnvOK'
#69BE5h   ''prow'
#69BF6h   ''pcol'
#69C07h   ''cursy'
#69C1Ah   ''cursx'
#69C2Dh   ''ttt'
#69C3Ch   ''source'
#69C51h   ''ofs'
#69C60h   ''tok'
#69C6Fh   ''rbv'
#69C7Eh   ''idfflg'
#69C93h   ''tmpop'
#69CA6h   ''tmppdat'
#69CBDh   ''ploc'
#69CCEh   ''bv'
#69CDBh   ''unbound'
```

E. Error Messages

Excluding any errors in supplementary libraries, this is the complete list of error messages that the HP 48 will display. They are listed by order of their code, given in both decimal and hexadecimal.

```
1    # 001h "Insufficient Memory"
2    # 002h "Directory Recursion"
3    # 003h "Undefined Local Name"
4    # 004h "Undefined XLIB Name"
5    # 005h "Memory Clear"
6    # 006h "Power Lost"
7    # 007h "Warning:"
8    # 008h "Invalid Card Data"
9    # 009h "Object In Use"
10   # 00Ah "Port Not Available"
11   # 00Bh "No Room in Port"
12   # 00Ch "Object Not in Port"
13   # 00Dh "Recovering Memory"
14   # 00Eh "Try To Recover Memory?"
15   # 00Fh "Replace RAM, Press ON
16   # 010h "No Mem To Config All"

257  # 101h "No Room to Save Stack"
258  # 102h "Can't Edit Null Char."
259  # 103h "Invalid User
                Function"
260  # 104h "No Current Equation"

262  # 106h "Invalid Syntax"
263  # 107h "Real Number"
264  # 108h "Complex Number"
265  # 109h "String"
266  # 10Ah "Real Array"
267  # 10Bh "Complex Array"
268  # 10Ch "List"
269  # 10Dh "Global Name"
270  # 10Eh "Local Name"
271  # 10Fh "Program"
272  # 110h "Algebraic"
273  # 111h "Binary Integer"
274  # 112h "Graphic"
275  # 113h "Tagged"
276  # 114h "Unit"
277  # 115h "XLIB Name"
278  # 116h "Directory"
279  # 117h "Library"
280  # 118h "Backup"
281  # 119h "Function"
282  # 11Ah "Command"
283  # 11Bh "System Binary"
284  # 11Ch "Long Real"
285  # 11Dh "Long Complex"
286  # 11Eh "Linked Array"
287  # 11Fh "Character"
288  # 120h "Code"
289  # 121h "Library Data"
290  # 122h "External"

292  # 124h "LAST STACK Disabled"
293  # 125h "LAST CMD Disabled"
294  # 126h "HALT Not Allowed"
295  # 127h "Array"
296  # 128h "Wrong Argument Count"
297  # 129h "Circular Reference"
298  # 12Ah "Directory Not Allowed"
299  # 12Bh "Non-Empty Directory"
300  # 12Ch "Invalid Definition"
301  # 12Dh "Missing Library"
302  # 12Eh "Invalid PPAR"
303  # 12Fh "Non-Real Result"
304  # 130h "Unable to Isolate"
305  # 131h "No Room to Show Stack"
306  # 132h "Warning:"
307  # 133h "Error:"
308  # 134h "Purge?"
309  # 135h "Out of Memory"
310  # 136h "Stack"
311  # 137h "Last Stack"
312  # 138h "Last Commands"
313  # 139h "Key Assignments"
314  # 13Ah "Alarms"
315  # 13Bh "Last Arguments"
316  # 13Ch "Name Conflict"
317  # 13Dh "Command Line"
```

```
513  # 201h "Too Few Arguments"
514  # 202h "Bad Argument Type"
515  # 203h "Bad Argument Value"
516  # 204h "Undefined Name"
517  # 205h "LASTARG Disabled"
518  # 206h "Incomplete
                    Subexpression"
519  # 207h "Implicit () off"
520  # 208h "Implicit () on"

769  # 301h "Positive Underflow"
770  # 302h "Negative Underflow"
771  # 303h "Overflow"
772  # 304h "Undefined Result"
773  # 305h "Infinite Result"

1281 # 501h "Invalid Dimension"
1282 # 502h "Invalid Array Element
1283 # 503h "Deleting Row"
1284 # 504h "Deleting Column"
1285 # 505h "Inserting Row"
1286 # 506h "Inserting Column"

1537 # 601h "Invalid ΣData"
1538 # 602h "Nonexistent ΣDAT"
1539 # 603h "Insufficient ΣData"
1540 # 604h "Invalid ΣPAR"
1541 # 605h "Invalid ΣData LN(Neg)"
1542 # 606h "Invalid ΣData LN(0)"
1543 # 607h "Invalid EQ"
1544 # 608h "Current equation:"
1545 # 609h "No current equation."
1546 # 60Ah "Enter eqn, press NEW"
1547 # 60Bh "Name the equation,
             press ENTER"
1548 # 60Ch "Select plot type"
1549 # 60Dh "Empty catalog"
1550 # 60Eh "undefined"
1551 # 60Fh "No stat data to plot"
1552 # 610h "Autoscaling"
1553 # 611h "Solving for "
1554 # 612h "No current data.
             Enter"
1555 # 613h "data point, press Σ+"
1556 # 614h "Select a model"
1557 # 615h "No alarms pending."
1558 # 616h "Press ALRM to create"
1559 # 617h "Next alarm:"
1560 # 618h "Past due alarm:"
1561 # 619h "Acknowledged"
1562 # 61Ah "Enter alarm, press
             SET"
1563 # 61Bh "Select repeat
             interval"
1564 # 61Ch "    I/O setup menu"
1565 # 61Dh "Plot type: "
1566 # 61Eh """"
1567 # 61Fh " (OFF SCREEN)"
1568 # 620h "Invalid PTYPE"
1569 # 621h "Name the stat data,
             press ENTER"
1570 # 622h "Enter value (zoom out if
             >1), press ENTER"
1571 # 623h "Copied to stack"
1572 # 624h "x axis zoom w/AUTO."
1573 # 625h "x axis zoom."
1574 # 626h "y axis zoom."
1575 # 627h "x and y axis zoom."
1576 # 628h "IR/wire:          "
1577 # 629h "ASCII/binary:     "
1578 # 62Ah "baud:             "
1579 # 62Bh "parity:           "
1580 # 62Ch "checksum type:  "
1581 # 62Dh "translate code:"
1582 # 62Eh "Enter matrix,
             then NEW"
```

2561 # A01h "Bad Guess(es)"
2562 # A02h "Constant?"
2563 # A03h "Interrupted"
2564 # A04h "Zero"
2565 # A05h "Sign Reversal"
2566 # A06h "Extremum"
2567 # B01h "Invalid Unit"
2568 # B02h "Inconsistent Units"

3073 # C01h "Bad Packet Block Check"
3074 # C02h "Timeout"
3075 # C03h "Receive Error"
3076 # C04h "Receive Buffer Overrun"
3077 # C05h "Parity Error"
3078 # C06h "Transfer Failed"
3079 # C07h "Protocol Error"
3080 # C08h "Invalid Server Cmd."
3081 # C09h "Port Closed"
3082 # C0Ah "Connecting"
3083 # C0Bh "Retry #"
3084 # C0Ch "Awaiting Server Cmd."
3085 # C0Dh "Sending "
3086 # C0Eh "Receiving "
3087 # C0Fh "Object Discarded"
3088 # C10h "Packet #"
3089 # C11h "Processing Command"
3090 # C12h "Invalid IOPAR"
3091 # C13h "Invalid PRTPAR"
3092 # C14h "Low Battery"
3093 # C15h "Empty Stack"
3094 # C16h "Row "
3095 # C17h "Invalid Name"

3329 # D01h "Invalid Date"
3330 # D02h "Invalid Time"
3331 # D03h "Invalid Repeat"
3332 # D04h "Nonexistent Alarm"

458752 # 70000h Last user message (message DOERR)

F. Machine Language Instructions

The instructions on the following pages are given in order of their codes. The HP HDS manual gives them in alphabetical order, but they are given here by code value, to make it easier to disassemble machine language programs (especially those in ROM). To make it even easier, the entire instruction set is given on two pages next to each other so that you won't even need to turn the page. More detailed explanations for these instructions are found in **Chapters 9** and **10**.

For the registers and fields, here is a summary of what we have already seen:

<table>
<tr><td>F</td><td>E</td><td>D</td><td>C</td><td>B</td><td>A</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">W</td></tr>
<tr><td>S</td><td colspan="12">M</td><td>XS</td><td colspan="2">B</td></tr>
<tr><td colspan="11"></td><td colspan="5">A</td></tr>
<tr><td colspan="13"></td><td colspan="3">X</td></tr>
</table>

Field	a	f	b
P	0	0	8
WP	1	1	9
XS	2	2	A
X	3	3	B
S	4	4	C
M	5	5	D
B	6	6	E
W	7	7	F
A		F	

00	RTNSXM	
01	RTN	
02	RTNSC	
03	RTNCC	
04	SETHEX	
05	SETDEC	
06	RSTK=C	
07	C=RSTK	
08	CLRST	
09	C=ST	
0A	ST=C	
0B	CSTEX	
0C	P=P+1	
0D	P=P-1	
0Ef0	A=A&B	f
0Ef1	B=B&C	f
0Ef2	C=C&A	f
0Ef3	D=D&C	f
0Ef4	B=B&A	f
0Ef5	C=C&B	f
0Ef6	A=A&C	f
0Ef7	C=C&D	f
0Ef8	A=A!B	f
0Ef9	B=B!C	f
0EfA	C=C!A	f
0EfB	D=D!C	f
0EfC	B=B!A	f
0EfD	C=C!B	f
0EfE	A=A!C	f
0EfF	C=C!D	f
0F	RTI	
100	R0=A	
101	R1=A	
102	R2=A	
103	R3=A	
104	R4=A	
108	R0=C	
109	R1=C	
10A	R2=C	
10B	R3=C	
10C	R4=C	
110	A=R0	
111	A=R1	
112	A=R2	
113	A=R3	
114	A=R4	
118	C=R0	
119	C=R1	
11A	C=R2	
11B	C=R3	
11C	C=R4	
120	AR0EX	
121	AR1EX	
122	AR2EX	
123	AR3EX	
124	AR4EX	
128	CR0EX	
129	CR1EX	
12A	CR2EX	
12B	CR3EX	
12C	CR4EX	
130	D0=A	
131	D1=A	
132	AD0EX	
133	AD1EX	
134	D0=C	
135	D1=C	
136	CD0EX	
137	CD1EX	
138	D0=AS	
139	D1=AS	
13A	AD0XS	
13B	AD1XS	

13C	D0=CS	
13D	D1=CS	
13E	CD0XS	
13F	CD1XS	
140	DAT0=A	A
141	DAT1=A	A
142	A=DAT0	A
143	A=DAT1	A
144	DAT0=C	A
145	DAT1=C	A
146	C=DAT0	A
147	C=DAT1	A
148	DAT0=A	B
149	DAT1=A	B
14A	A=DAT0	B
14B	A=DAT1	B
14C	DAT0=C	B
14D	DAT1=C	B
14E	C=DAT0	B
14F	C=DAT1	B
150a	DAT0=A	a
151a	DAT1=A	a
152a	A=DAT0	a
153a	A=DAT1	a
154a	DAT0=C	a
155a	DAT1=C	a
156a	C=DAT0	a
157a	C=DAT1	a
158x	DAT0=A	x+1
159x	DAT1=A	x+1
15Ax	A=DAT0	x+1
15Bx	A=DAT1	x+1
15Cx	DAT0=C	x+1
15Dx	DAT1=C	x+1
15Ex	C=DAT0	x+1
15Fx	C=DAT1	x+1
16n	D0=D0+	n+1
17n	D1=D1+	n+1
18n	D0=D0-	n+1
19pq	D0=(2)	qp
1Apqrs	D0=(4)	srqp
1Bpqrst	D0=(5)	tsrqp
1Cn	D1=D1-	n+1
1Dpq	D1=(2)	qp
1Epqrs	D1=(4)	srqp
1Fpqrst	D1=(5)	tsrqp
2n	P=	n
3xh0..hx	LCHEX	#hx..h0
400	RTNC	
420	NOP3	
4yz	GOC	zy
500	RTNNC	
5yz	GONC	zy
6300	NOP4	
64000	NOP5	
6yzt	GOTO	tzy
7yzt	GOSUB	tzy
800	OUT=CS	
801	OUT=C	
802	A=IN	
803	C=IN	
804	UNCNFG	
805	CONFIG	
806	C=ID	
807	SHUTDN	
8080	INTON	
80810	RSI	
8082xh0..hx	LAHEX	#hx..h0

8083	BUSCB	
8084d	ABIT=0	d
8085d	ABIT=1	d
8086d	?ABIT=0	d
8087d	?ABIT=1	d
8088d	CBIT=0	d
8089d	CBIT=1	d
808Ad	?CBIT=0	d
808Bd	?CBIT=1	d
808C	PC=(A)	
808D	BUSCD	
808E	PC=(C)	
808F	INTOFF	
809	C+P+1	
80A	RESET	
80B	BUSCC	
80Cx	C=P	x
80Dx	P=C	x
80E	SREQ?	
80Fx	CPEX	x
810	ASLC	
811	BSLC	
812	CSLC	
813	DSLC	
814	ASRC	
815	BSRC	
816	CSRC	
817	DSRC	
818f0x	A=A+x+1	f
818f1x	B=B+x+1	f
818f2x	C=C+x+1	f
818f3x	D=D+x+1	f
818f8x	A=A-(x+1)	f
818f9x	B=B-(x+1)	f
818fAx	C=C-(x+1)	f
818fBx	D=D-(x+1)	f
819f0	ASRB	f
819f1	BSRB	f
819f2	CSRB	f
819f3	DSRB	f
81Af00	R0=A	f
81Af01	R1=A	f
81Af02	R2=A	f
81Af03	R3=A	f
81Af04	R4=A	f
81Af08	R0=C	f
81Af09	R1=C	f
81Af0A	R2=C	f
81Af0B	R3=C	f
81Af0C	R4=C	f
81Af10	A=R0	f
81Af11	A=R1	f
81Af12	A=R2	f
81Af13	A=R3	f
81Af14	A=R4	f
81Af18	C=R0	f
81Af19	C=R1	f
81Af1A	C=R2	f
81Af1B	C=R3	f
81Af1C	C=R4	f
81Af20	AR0EX	f
81Af21	AR1EX	f
81Af22	AR2EX	f
81Af23	AR3EX	f
81Af24	AR4EX	f
81Af28	CR0EX	f
81Af29	CR1EX	f
81Af2A	CR2EX	f
81Af2B	CR3EX	f
81Af2C	CR4EX	f
81B2	PC=A	
81B3	PC=C	
81B4	A=PC	
81B5	C=PC	

81B6	APCEX	
81B7	CPCEX	
81C	ASRB	
81D	BSRB	
81E	CSRB	
81F	DSRB	
821	XM=0	
822	SB=0	
824	SR=0	
828	MP=0	
82F	CLRHST	
831	?XM=0	
832	?SB=0	
834	?SR=0	
838	?MP=0	
84d	ST=0	d
85d	ST=1	d
86d	?ST=0	d
87d	?ST=1	d
88n	?P≠	n
89n	?P=	n
8A0	?B=A	A
8A1	?C=B	A
8A2	?A=C	A
8A3	?C=D	A
8A4	?B≠A	A
8A5	?C≠B	A
8A6	?A≠C	A
8A7	?D≠C	A
8A8	?A=0	A
8A9	?B=0	A
8AA	?C=0	A
8AB	?D=0	A
8AC	?A≠0	A
8AD	?B≠0	A
8AE	?C≠0	A
8AF	?D≠0	A
8B0	?A>B	A
8B1	?B>C	A
8B2	?C>A	A
8B3	?D>C	A
8B4	?A<B	A
8B5	?B<C	A
8B6	?C<A	A
8B7	?D<C	A
8B8	?A≥B	A
8B9	?B≥C	A
8BA	?C≥A	A
8BB	?D≥C	A
8BC	?A≤B	A
8BD	?B≤C	A
8BE	?C≤A	A
8BF	?D≤C	A
8Cpqrs	GOLONG	srqp
8Dpqrst	GOVLNG	tsrqp
8Epqrs	GOSUBL	srqp
8Fpqrst	GOSBVL	tsrqp
9a0	?A=B	a
9a1	?B=C	a
9a2	?C=A	a
9a3	?C=D	a
9a4	?A≠B	a
9a5	?B≠C	a
9a6	?C≠A	a
9a7	?D≠C	a
9a8	?A=0	a
9a9	?B=0	a
9aA	?C=0	a
9aB	?D=0	a
9aC	?A≠0	a
9aD	?B≠0	a
9aE	?C≠0	a
9aF	?D≠0	a
9b0	?A>B	b
9b1	?B>C	b
9b2	?C>A	b
9b3	?D>C	b
9b4	?A<B	b
9b5	?B<C	b
9b6	?C<A	b
9b7	?D<C	b
9b8	?A≥B	b
9b9	?B≥C	b
9bA	?C≥A	b
9bB	?D≥C	b
9bC	?A≤B	b
9bD	?B≤C	b
9bE	?C≤A	b
9bF	?D≤C	b
Aa0	A=A+B	a
Aa1	B=B+C	a
Aa2	C=C+A	a
Aa3	D=D+C	a
Aa4	A=A+A	a
Aa5	B=B+B	a
Aa6	C=C+C	a
Aa7	D=D+D	a
Aa8	B=B+A	a
Aa9	C=C+B	a
AaA	A=A+C	a
AaB	C=C+D	a
AaC	A=A-1	a
AaD	B=B-1	a
AaE	C=C-1	a
AaF	D=D-1	a
Ab0	A=0	b
Ab1	B=0	b
Ab2	C=0	b
Ab3	D=0	b
Ab4	A=B	b
Ab5	B=C	b
Ab6	C=A	b
Ab7	D=C	b
Ab8	B=A	b
Ab9	C=B	b
AbA	A=C	b
AbB	C=D	b
AbC	ABEX	b
AbD	BCEX	b
AbE	ACEX	b
AbF	CDEX	b
Ba0	A=A-B	a
Ba1	B=B-C	a
Ba2	C=C-A	a
Ba3	D=D-C	a
Ba4	A=A+1	a
Ba5	B=B+1	a
Ba6	C=C+1	a
Ba7	D=D+1	a
Ba8	B=B-A	a
Ba9	C=C-B	a
BaA	A=A-C	a
BaB	C=C-D	a
BaC	A=B-A	a
BaD	B=C-B	a
BaE	C=A-C	a
BaF	D=C-D	a
Bb0	ASL	b
Bb1	BSL	b
Bb2	CSL	b
Bb3	DSL	b
Bb4	ASR	b
Bb5	BSR	b
Bb6	CSR	b
Bb7	DSR	b
Bb8	A=-A	b
Bb9	B=-B	b
BbA	C=-C	b
BbB	D=-D	b
BbC	A=-A-1	b
BbD	B=-B-1	b
BbE	C=-C-1	b
BbF	D=-D-1	b
C0	A=A+B	A
C1	B=B+C	A
C2	C=C+A	A
C3	D=D+C	A
C4	A=A+A	A
C5	B=B+B	A
C6	C=C+C	A
C7	D=D+D	A
C8	B=B+A	A
C9	C=C+B	A
CA	A=A+C	A
CB	C=C+D	A
CC	A=A-1	A
CD	B=B-1	A
CE	C=C-1	A
CF	D=D-1	A
D0	A=0	A
D1	B=0	A
D2	C=0	A
D3	D=0	A
D4	A=B	A
D5	B=C	A
D6	C=A	A
D7	D=C	A
D8	B=A	A
D9	C=B	A
DA	A=C	A
DB	C=D	A
DC	ABEX	A
DD	BCEX	A
DE	ACEX	A
DF	CDEX	A
E0	A=A-B	A
E1	B=B-C	A
E2	C=C-A	A
E3	D=D-C	A
E4	A=A+1	A
E5	B=B+1	A
E6	C=C+1	A
E7	D=D+1	A
E8	B=B-A	A
E9	C=C-B	A
EA	A=A-C	A
EB	C=C-D	A
EC	A=B-A	A
ED	B=C-B	A
EE	C=A-C	A
EF	D=C-D	A
F0	ASL	A
F1	BSL	A
F2	CSL	A
F3	DSL	A
F4	ASR	A
F5	BSR	A
F6	CSR	A
F7	DSR	A
F8	A=-A	A
F9	B=-B	A
FA	C=-C	A
FB	D=-D	A
FC	A=-A-1	A
FD	B=-B-1	A
FE	C=-C-1	A
FF	D=-D-1	A

G. Glossary

Address A number between 0 and FFFFF (in hexadecimal) which indicates the location in memory of some data.

Annunciator One of the symbols that appear in the status area (the very top of the HP48 calculator) to indicate the machine's current status (**DEG**, **RAD**, **GRAD**, α, ⧖, etc.).

Assemble The act of translating an assembly program into machine language.

Assembler A program that will translate an assembly program into machine language.

Bank-switching A technique used to have two distinct memory areas exist at the same address. One of the two is visible, while the other is hidden. To access the hidden memory, the visible memory must be moved to another address.

BCD (**B**inary **C**oded **D**ecimal) This is a method of storing a decimal number in binary. For example, the number 20 (in decimal) would be stored as 20h (in hexadecimal) which actually equals 32 (in decimal).

Bit A memory location that can equal 0 or 1. This is the basic unit that makes up a nibble.

Bit clear To say that a bit is clear means that it equals zero.

Bit set To say that a bit is set means that it equals one.

Buffer A memory area that is used as a temporary storage for information that is waiting to be used. For example, each keypress is stored in a buf fer, and the data going out or coming in the RS232c port goes through a buf fer.

Byte 8 bits of data. The basic unit of measurement for memory size. A byte can represent any value from 0 to 255 (decimal) or from 0 to FF (hexadecimal).

Disassemble Translate a machine language program into assembly.

Disassembler A program that will translate a machine language program into assembly.

Field A part of a register.

Flag One bit in memory that serves as an indicator.

Garbage Collector This operation is performed when the machine does not have enough free memory to perform an operation. This operation consists of purging any temporary objects that are no longer being used. The MEM command will cause garbage collection to occur.

Hexadecimal Base 16. The digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Kilobyte (Kb) 1024 (2^{10}) bytes. A unit of measurement for memory size.

LCD (**L**iquid **C**rystal **D**isplay) The HP48 screen is an LCD screen.

Machine Language A list of codes which represent elementary instructions that the microprocessor is capable of understanding.

Memory A place used for storing data. See **RAM** and **ROM**.

Nibble 4 bits of data. This is the basic unit if memory for the HP 48 calculator. A nibble can represent any value from 0 to 15 (decimal) or from 0 to F (hexadecimal).

Object Everything that RPL can handle is called an object. A real number, for example, is an object.

Peek A program (or instruction) that will read the contents of a specific memory location.

Poke A program (or instruction) that will write data to a specific memory location.

Processor See microprocessor.

Prolog A group of 5 nibbles which serve as an object's identification. The prolog is always the first 5 nibbles of an object.

RAM (**R**andom **A**ccess **M**emory) RAM consists of electronic circuits that are capable of storing data. RAM can be modified.

Register A memory location for the microprocessor. Typically faster access than RAM, so most operations are performed in registers. Registers can contain only positive integers.

ROM (**R**ead **O**nly **M**emory) ROM consists of electronic circuits that are capable of storing data. ROM cannot be modified. ROM contains the machine language instructions for RPL, among other things.

RS-232C A data communications method used by the HP 48 to transfer information between itself and another computer. The data is sent serially—one bit at a time.

Stack The stack is a method of temporary storage. The user stack is displayed in the central part of the HP 48 screen. RPL is based on the principle of the stack.

H. Handy Machine Language Routines

Here are a few machine language routines found in ROM that will perform useful functions to add to your machine language programs. They should generally be called with a GOSBVL.

- SAVE_REG (# 0679Bh) will backup the registers **D0**, **D1**, **B**, field **A**, and **D**, field **A** into a specific memory area (see **Part 2, Chapter 7**). Note that they are not saved on a stack, so if you call this routine a second time, the first values are lost.
- LOAD_REG (# 067D2h) restores the values saved by SAVE_REG.
- TRDN (# 0670Ch) copies C, field A nibbles pointed to by D0 to the address in D1 (beginning addresses of two memory areas). D1 should be less than D0 for this routine (transfer down).
- TRUP (# 066B9h) copies C, field A nibbles pointed to by D0 to the address in D1 (ending addresses of two memory areas). D1 should be less than D0 for this routine (transfer up).
- ZEROM (# 0675Ch) sets C, field A nibbles pointed to by D1 to zero.
- RES_ROOM (# 039BEh) reserves C, field A nibbles of RAM. The address of the reserved area is stored in D0. If the free memory is not sufficient, then a garbage collection will occur. If this does not free enough memory, then the program will halt, and an error message will be displayed.
- GARB_COLL (# 0613Eh) cleans the HP48 memory by purging all unused objects (unreferenced objects found in temporary RAM).
- RES_STR (# 05B7Dh) reserves a string of characters of length (in nibbles) C, field A. This routine returns the address of the string in R0, field A and the address of its contents in D0. If the free memory is not sufficient, then a garbage collection will occur. If this does not free enough memory, then the program will halt, and an error message will be displayed.
- PUSH_R0 (# 06537h) places the value of R0, field A onto the stack as a system binary. **CAUTION:** The registers D1 and D must have been previously saved with a call to SAVE_REG.

- PUSH_R0_R1 (# 06529h) places the values of **R0**, field A and **R1**, field A onto the stack as system binaries. R0 will be in level 2, and R1 will be in level 1. CAUTION: The registers D1 and D must be saved previously with a call to SAVE_REG.
- POP_C (# 06641h) takes the value of a system binary from the stack and puts it in C, field A. **CAUTION:** The registers D1 and D must be the system values (stack pointer and free memory). Their values will be modified by POP_C (since the object in level 1 was removed).
- POP_C_A (# 03F5Dh) takes the values of two system binaries from the stack. As with the routine above, D1 and D are modified. C, field A will contain the number from level 1, and A, field A will contain the number from level 2.
- DIV5 (# 06A8Eh) divides the contents of C, field A by 5. This routine uses the first 10 nibbles of registers A, C, and D. This actually performs a multiplication by 3355444, then a division by 16777216, which is just about a division by 5.
- MULTA (# 03991h) multiplies A, field A and C, field A, and puts the result in B, field A.
- BEEP (# 017A6h) emits a sound with a frequency of D, field A and a duration in milliseconds of C, field A. This routine takes into account flag -56.
- ERROR (# 05023h) displays the error message for the number contained in A, field **A. CAUTION:** This routine must be called with a GOTO, and not a GOSUB. It will halt the program currently executing. This call must be preceded by a call to LOAD_REG, if you have called SAVE_REG.
- STOP (# 10FDBh) called with a GOTO, will halt the program currently executing. It generates error #123h which IFERR cannot handle, so the calculator is returned to interactive mode. This call must be preceded by a call to LOAD_REG, if you have called SAVE_REG.
- EXHR (# 026BFh) will execute the routine in hidden ROM at the address contained in C, field A.

- DIV (# 65807h) divides A, field W by C, field W. The result is placed in field W of both registers A and C, and the remainder is placed in B, field W.
- MULT (# 53EE4h) multiplies A, field W and C, field W. The result is placed in field W of both registers A, and C.
- FREEMEM (# 069F7h) recalculates the value in #7066Eh (free memory in 5 nibble blocks) using the addresses in #70579h and #70574h. This call should only be used if you have previously called SAVE_REG (which you would typically do at the beginning of your program).
- FREEMEMQ (# 06806h) calculates the amount of free memory in nibbles. The result is placed in C, field A. This call should only be used if you have previously called SAVE_REG.
- ASLW5 (# 0D5F6h) executes the function ASL on field W 5 times, which helps you use one register as three fields of 5 nibbles (when used in conjunction with ASLW5).
- ASRW5 (# 0D5E5h) executes the function ASR on field W 5 times.
- CSLW5 (# 0D618h) executes the function CSL on field W 5 times.
- CSRW5 (# 0D607h) executes the function CSR on field W 5 times.
- D07FMAP (# 0C1B0h) stores in nibble #4 of D0, the base address of built-in RAM (7 or F).
- D17FMAP (# 0C1A1h) stores in nibble #4 of D1, the base address of built-in RAM (7 or F).
- D17FMAP2 (# 0C154h) stores in nibble #4 of D1, the base address of built-in RAM (7 or F). This routine is slower that D17FMAP, but it modifies only nibble #4, and the others are left unchanged.
- CONFTABCRC (# 09B73h) calculates the checksum for the configuration table at #7042Ch. The result is placed in C, field A.

- CHECK_BAT (# 006EDh) checks the batteries, depending on the value in nibble #0 of C: 1 to test if the main batteries are very weak, 2 to test if the main batteries are weak, 4 to test the battery for the plug-in card in port 1, and 8 to test the batter for the plug-in card in port 2. On return, the CARRY is set if the battery is weak.
- CHECK_BATI (# 325AAh) checks the main batteries. If the batteries are weak, the CARRY is set, and the corresponding error number is placed in C, field A.
- D0TOS (# 6384Eh) places the address stored in #70579h (the address of the object in stack level 1) into D0. SAVE_REG must have been called previously.
- D1TOS (# 6385Dh) places the address stored in #70579h (the address of the object in stack level 1) into **D1**. SAVE_REG must have been called previously.
- DISINTR (# 01115h) disables interrupts.
- ALLINTR (# 010E5h) enables interrupts.
- DISPOFF (# 01BBDh) turns off the display.
- ADISPOFF (# 01BD3h) turns off the display and the annunciators.
- DISPON (# 01B8Fh) turns on the display.
- ADISPON (# 01BA5h) turns on the display and the annunciators.
- EMPTKBUF (# 00D57h) clears the keyboard buffer.
- EMPTATTN (# 00D8Eh) sets the five nibbles at #70679h to zero. (This is the area that stores how many times the ON key has been pressed).
- KEYINBUFF (#04999h) tests the keybuffer for keys that have been pressed. On return the CARRY is clear if the buffer is empty.

- DISPINGROB (# 11D8Fh) writes text into a graphics object using the 5x7 font. It takes the address of the text beginning in **D1,** the address of where to write into the GROB in D0, the number of characters to write in C, field A, the left margin (in characters) in B, field A, and the size (in nibbles) of the GROB in D, field A. **CAUTION:** This size is the total size of the GROB, and can be calculated by finding the integer part of [((size in pixels) + 7) / 4].
- IR7CONF (# 026E6h) configures the built-in RAM to the address #70000h. This routine updates the graphics pointers.
- IRFCONF (# 0228Eh) configures the built-in RAM to #F0000h. Do displace the built-in RAM to this address, first unconfigure it, then call IRFCONF, then CONFGRAPH.
- CONFGRAPH (# 01C7Fh) recalculates the graphics pointers after displacing the built-in RAM.
- BUSYON (# 42333h) turns on the BUSY annunciator.
- BUSYNO (# 42359h) turns off the BUSY annunciator.

I. Index

The most advanced book about machine language programming on the HP48 s/sx series. With this book you'll be able to get the best of your favourite calculator !

Translated from the French book « Voyage au centre de la HP48 s/sx » by Douglas R. Cannon

ISBN 9789995911959

90000

9 789995 911959