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

Hewlett-Packard, HP-71, HP-28, HP 48, HP 48S, HP 48SX, Macintosh, Atari, UNIX, Amiga and IBM are registered tradenames or trademarks.

© 1993, Paul Courbis and Sébastien Lalande. All rights reserved. No portion of this book or its contents, nor any portion of the programs contained herein, may be reproduced in any form, printed, electronic or mechanical, without written permission from Paul Courbis, Sébastien Lalande, and Grapevine Publications, Inc.

Printed in the United States of America

First Printing – December, 1993

Notice of Disclaimer: The authors and Grapevine Publications, Inc. make no express or implied warranty with regard to the keystroke procedures and program materials herein offered, nor to their merchantability nor fitness for any particular purpose. These keystroke procedures and program materials are made available solely on an "as is" basis, and the entire risk as to their quality and performance is with the user. Should the keystroke procedures and program materials prove defective, the user (and not Grapevine Publications, Inc., nor any other party) shall bear the entire cost of all necessary correction and all incidental or consequential damages. Grapevine Publications, Inc. shall not be liable for any incidental or consequential damages in connection with, or arising out of, the furnishing, use, or performance of these keystroke procedures or program materials.
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.

Introduction .................................................................................................................. 12

1. First Approach to the HP 48 ................................................................. 14
   Getting started and finding your way through
   the maze of inscriptions on the HP 48 keyboard.

2. Reverse Polish Notation .......................................................................... 18
   Basic principles of RPN, with examples and exercises.

3. Organizing Your Data Properly ............................................................... 28
   How to use directory trees to store data
   in an easily retrievable manner.

4. Programming the HP 48 ........................................................................... 34
   What a program is and how to write one;
   how the HP 48 programming language works;
   programming advice and step-by-step examples.

5. Presenting Your Data Properly ................................................................. 46
   How to present your programs and data in a
   user-friendly manner; the CST menu; key redefinitions.

6. Saving and Transmitting Data ................................................................. 52
   Taking advantage of the HP 48's ability to exchange data
   with the outside world: memory cards, the RS-232C port,
   the infrared receiver/transmitter.

7. Other Strong Points of the HP 48 ........................................................... 58
   Several incredible tools: symbolic calculation, graphics,
   units management, and more.

Conclusion .................................................................................................................. 62
Part Two: Machine Language

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

Introduction ..................................................................................................................... 64

8. Machine Language ....................................................................................................... 68
   An initiation to machine language; basic tools and useful concepts for understanding the rest of this section.

9. The Saturn Microprocessor ......................................................................................... 72
   A general view of the HP 48's microprocessor; a detailed view of all its registers and their unique roles.

10. The Saturn Instruction Set ......................................................................................... 82
    All the available instructions, classified by function type and by registers used.

11. HP 48 Objects ........................................................................................................... 122
    Principles of memory storage for all objects accessible to the user (real numbers, binary integers, graphic objects, and others).

12. General Memory Organization ..................................................................................... 158
    A global view of HP 48 memory to prepare for the detailed explanations that follow.

13. I/O RAM .................................................................................................................... 162
    How to directly access certain HP 48 peripherals (the clock, infrared I/O, etc.).

14. RAM ........................................................................................................................... 174
    A detailed explanation of the HP 48's RAM organization.

15. Programming in Machine Language ............................................................................ 206
    How to access all resources of the HP 48.
# Part Three: Library of Programs

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

---

### Notice

How to type in a machine language program.

### Programs dealing with Machine Language

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASS</td>
<td>Installing assembly language programs</td>
<td>215</td>
</tr>
<tr>
<td>ALLBYTES</td>
<td>Calculate all checksums in a directory</td>
<td>216</td>
</tr>
<tr>
<td>BY5</td>
<td>Display code strings in a readable form</td>
<td>217</td>
</tr>
<tr>
<td>CLEAN</td>
<td>Cleanup of code strings</td>
<td>218</td>
</tr>
<tr>
<td>PEEK</td>
<td>Read from HP 48 memory</td>
<td>220</td>
</tr>
<tr>
<td>POKE</td>
<td>Write to HP 48 memory</td>
<td>222</td>
</tr>
<tr>
<td>HRPEEK</td>
<td>Read from the HP 48 hidden ROM</td>
<td>224</td>
</tr>
<tr>
<td>?ADR</td>
<td>Determine the address of a stack object</td>
<td>228</td>
</tr>
<tr>
<td>SSAG</td>
<td>Inverse of GASS</td>
<td>229</td>
</tr>
<tr>
<td>RASS</td>
<td>A faster version of GASS</td>
<td>230</td>
</tr>
<tr>
<td>CHK</td>
<td>An argument verifier</td>
<td>232</td>
</tr>
<tr>
<td>REVERSE</td>
<td>Reverse strings</td>
<td>236</td>
</tr>
<tr>
<td>CRNAME</td>
<td>Create non-standard names</td>
<td>238</td>
</tr>
<tr>
<td>CLVAR</td>
<td>Remove the CLVAR function</td>
<td>239</td>
</tr>
<tr>
<td>SYSEVAL</td>
<td>Remove the SYSEVAL function</td>
<td>240</td>
</tr>
<tr>
<td>CONTRAST</td>
<td>Adjust the contrast from a program</td>
<td>241</td>
</tr>
<tr>
<td>DISPOFF and DISPON</td>
<td>Turn off and on the display</td>
<td>241</td>
</tr>
<tr>
<td>FAST</td>
<td>Speeding up the HP 48</td>
<td>242</td>
</tr>
<tr>
<td>DISASM</td>
<td>A SATURN disassembler</td>
<td>243</td>
</tr>
<tr>
<td>B→SB</td>
<td>Binary integer to system binary</td>
<td>258</td>
</tr>
<tr>
<td>SB→B</td>
<td>System binary to binary integer</td>
<td>258</td>
</tr>
<tr>
<td>R→SB</td>
<td>Real number to system binary</td>
<td>258</td>
</tr>
<tr>
<td>SB→R</td>
<td>System binary to real number</td>
<td>258</td>
</tr>
<tr>
<td>C→SB</td>
<td>Character to system binary</td>
<td>258</td>
</tr>
<tr>
<td>SB→C</td>
<td>System binary to character</td>
<td>258</td>
</tr>
<tr>
<td>ROMRCL</td>
<td>Recall objects in hidden ROM</td>
<td>259</td>
</tr>
<tr>
<td>A→STR and STR→A</td>
<td>Convert a string from and to an address</td>
<td>260</td>
</tr>
<tr>
<td>BFREE</td>
<td>Find free space on RAM card in BACKUP mode</td>
<td>261</td>
</tr>
<tr>
<td>SEARCH</td>
<td>A subroutine for the other SEARCH programs</td>
<td>262</td>
</tr>
</tbody>
</table>
ROMSEARCH  Find an object in ROM ........................................... 263
RAMSEARCH  Find an object in RAM ........................................... 263
MODUSEARCH  Find an object in a plug-in card ......................... 264
CRC       Calculate the checksum ........................................... 265
CRCLM     An assembly version of CRC ..................................... 265

Mathematical Programs

CALC      An infinite precision, integer calculator ...................... 266
PI         Calculate $\pi$ to any precision .............................. 286
VAL       Value of a polynomial stored as a vector ................... 288
DER       Solve a polynomial stored as a vector ....................... 288
A$\rightarrow$V and V$\rightarrow$A  Convert algebraic polynomial to/from a vector ... 289
DIVP      Division of two polynomials as vectors ...................... 290
PCAR      Calculation of characteristic polynomials .................. 291
LAGU      Universal polynomial root finder ........................... 292
PMAT      Multiplying a matrix by a polynomial ...................... 294
mSOLVER   Solving systems of equations ............................... 295

Games

MAZE      Escape from the cursed maze! .................................. 298
MASTER    Mastermind ....................................................... 304
ANAG      Find all the anagrams of a word ............................ 307
SQUARE    Magic Square .................................................... 308

Miscellaneous Programs

PR40      Print in 40 columns ............................................ 311
DSP and INITSCR A 33-column text display ............................ 312
MUSICLM   A little music .................................................. 314
MODUL     Sound effects .................................................. 316
RABIP     Random music ................................................... 318
JINGLE    Some friendly music ............................................ 318
RENAME    Rename a variable ............................................. 319
AUTOST    A Start-up program ............................................. 319
CAL       A calendar (one month display) .............................. 320
CIRCLE    Fast circle drawing ............................................ 322
BANNER    Display in giant letters ...................................... 324
### Appendices

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Answers to Exercises</td>
<td>332</td>
</tr>
<tr>
<td>B</td>
<td>Background Information</td>
<td>338</td>
</tr>
<tr>
<td></td>
<td>How to find out your machine's ROM version; what to do in case of a disaster; explanations of concepts dear to computer scientists: hexadecimal, binary, bits, nibbles, and bytes.</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>RPL Commands in alphabetical order</td>
<td>345</td>
</tr>
<tr>
<td></td>
<td>by instruction number</td>
<td>350</td>
</tr>
<tr>
<td>D</td>
<td>Objects in ROM</td>
<td>354</td>
</tr>
<tr>
<td></td>
<td>A list of objects already coded by Hewlett-Packard—why go to all the trouble when the work is already done?</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Error Messages</td>
<td>374</td>
</tr>
<tr>
<td></td>
<td>All the error messages that the HP 48 will ever display.</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Machine Language Instruction Set</td>
<td>378</td>
</tr>
<tr>
<td></td>
<td>In two pages, all the HP 48 assembly instructions with accompanying codes—ideal for the machine language programmer.</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>Glossary</td>
<td>382</td>
</tr>
<tr>
<td>H</td>
<td>Handy Machine Language Routines</td>
<td>386</td>
</tr>
<tr>
<td></td>
<td>A few ML programs found in ROM that are already done for you.</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Index</td>
<td>392</td>
</tr>
</tbody>
</table>
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 different 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 be 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 (orange) key is the (alpha) key. If you press (alpha) 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 (alpha) key, then that letter will appear on the screen. For example, pressing (alpha) 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 (alpha) twice. To exit this mode, simply press (alpha) once more. To type 'AB' you would press the buttons: ▼αα ▼AB 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 \( \leftarrow \text{MEMORY} \)), you would be placed in the memory menu:

```
MEM BYTES VARS ORDER PATH CDRIR
```

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: 123 [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—\(→\text{CLR}\)—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: |
  | 1: |
  | 2 |
  | 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:

\[
\begin{array}{c}
4: \\
3: \\
2: \\
1: \\
\end{array}
\]

\[
\begin{array}{c}
5 \\
\end{array}
\]

• \( 4 \text{ ENTER} \) shows:

\[
\begin{array}{c}
4: \\
3: \\
2: \\
1: \\
\end{array}
\]

\[
\begin{array}{c}
5 \\
4 \\
\end{array}
\]

• \( 5 \text{ ENTER} \) shows:

\[
\begin{array}{c}
4: \\
3: \\
2: \\
1: \\
\end{array}
\]

\[
\begin{array}{c}
5 \\
4.5 \\
5 \\
\end{array}
\]

• \( + \) gives:

\[
\begin{array}{c}
4: \\
3: \\
2: \\
1: \\
\end{array}
\]

\[
\begin{array}{c}
5 \\
9 \\
\end{array}
\]

• And finally, \( \times \) gives the result:

\[
\begin{array}{c}
4: \\
3: \\
2: \\
1: \\
\end{array}
\]

\[
\begin{array}{c}
45 \\
\end{array}
\]

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:

  Before:
  
  - 4:
  - 3:
  - 2:
  - 1:

  After `SWAP`:
  
  - 4:
  - 3:
  - 2:
  - 1:

- **DROP (DROP)** drops (erases) the object in level 1. For example:

  Before:
  
  - 4:
  - 3:
  - 2:
  - 1:

  After `DROP`:
  
  - 4:
  - 3:
  - 2:
  - 1:

- **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:

  ![Stack Diagram]

  After pressing **OVER**:

  ![Stack Diagram]

- **ROT** rotates the 3 first stack entries:

  ![Stack Diagram]

  After pressing **ROT**:

  ![Stack Diagram]

- **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:

<table>
<thead>
<tr>
<th>Level</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:</td>
<td>4</td>
</tr>
<tr>
<td>3:</td>
<td>5</td>
</tr>
<tr>
<td>2:</td>
<td>6</td>
</tr>
<tr>
<td>1:</td>
<td>3</td>
</tr>
</tbody>
</table>

After pressing **ROLLD**:

<table>
<thead>
<tr>
<th>Level</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:</td>
<td>6</td>
</tr>
<tr>
<td>3:</td>
<td>4</td>
</tr>
<tr>
<td>2:</td>
<td>5</td>
</tr>
<tr>
<td>1:</td>
<td></td>
</tr>
</tbody>
</table>

(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:

<table>
<thead>
<tr>
<th>Level</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:</td>
<td>123456789</td>
</tr>
<tr>
<td>3:</td>
<td>1</td>
</tr>
<tr>
<td>2:</td>
<td>3</td>
</tr>
<tr>
<td>1:</td>
<td></td>
</tr>
</tbody>
</table>

After pressing **PICK**:

<table>
<thead>
<tr>
<th>Level</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:</td>
<td>123456789</td>
</tr>
<tr>
<td>3:</td>
<td>1</td>
</tr>
<tr>
<td>2:</td>
<td>1</td>
</tr>
<tr>
<td>1:</td>
<td>123456789</td>
</tr>
</tbody>
</table>

(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:

<table>
<thead>
<tr>
<th>Level</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:</td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>33333</td>
</tr>
<tr>
<td>1:</td>
<td>44444</td>
</tr>
</tbody>
</table>
After pressing **DEPTH**:

<table>
<thead>
<tr>
<th>4:</th>
<th>3:</th>
<th>2:</th>
<th>1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>33333</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>44444</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

(there were 2 objects on the stack).

- **DUP** duplicates the object found in level 1:

<table>
<thead>
<tr>
<th>4:</th>
<th>3:</th>
<th>2:</th>
<th>1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

After pressing **DUP**:

- **DUP2** duplicates the first 2 objects of the stack:

<table>
<thead>
<tr>
<th>4:</th>
<th>3:</th>
<th>2:</th>
<th>1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

After pressing **DUP2**:

- **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**.

2. **Reverse Polish Notation**
• DROP2 “drops” the first 2 objects from the stack:

After pressing \textbf{DROP2}:

• 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 \( \frac{5}{(3+1) \cdot (9-5)} \)

2-2. If the stack contains:

\[
\begin{array}{c|c}
4: & 3: \\
3: & 2: \\
2: & 1: \\
1: & 1 \\
\end{array}
\]

how would you arrive at the following stack?

\[
\begin{array}{c|c}
4: & 3: \\
3: & 2: \\
2: & 1: \\
1: & 3 \\
\end{array}
\]

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

\[
5 \text{ ENTER } 3 \times 1 1 - 4 + 1 - \cos
\]

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, \texttt{\{MEMORY\}} (MEMORY) takes you to the MEMORY menu, which is a list of internal functions that provide memory management. Now, if you press \texttt{\{A\}} (the white button below \texttt{\{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 \texttt{\{A\}} button, the HP 48 knew that you wanted to execute the object \texttt{\{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 \texttt{\{NXT\}} (NEXT page) and \texttt{\{PREV\}} (PREVIOUS page). Thus if you were to press \texttt{\{NXT\}}, you would be able to use the other functions of the menu MEMORY (and if you continually press \texttt{\{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: \( \text{MODES} \) 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 ASC ASC
- **Page 4:** HEX DEC OCT BIN FM.

If you press \( \text{CLK} \) (found at the end of page 2), the time and date appear (or disappear) at the top of the screen, and the label \( \text{CLK} \) becomes \( \text{CLK} \). When a "\( \Box \)" 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 \( \text{PRG} \)). This means that if you press the corresponding button, you will enter a submenu 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 \texttt{TIME ((\texttt{\textsc{time}}))}, has this tree structure (partially represented here):

![Menu Tree Diagram]

Sub-menus can contain objects, or they can have their own sub-menus (for example \texttt{RPT} is a sub-menu of the sub-menu \texttt{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 \( \text{UPDIR} \); to return directly to the root, press \( \text{HOME} \). 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 \( \text{STOre} \). For example, press \( 5 1 2 \text{ENTER} \), which places the real number 512 on the stack. Then press \( \alpha \alpha \alpha \alpha \alpha \alpha \text{ENTER} \). The screen should show:

```
4:
3:
2: 912
1: 'ABC'
```

Now press \( \text{VAR} \), to place you in your working directory, then \( \text{STO} \), to store the number. \( \text{ABC} \) should appear to the left of the current directory menu.

To recall this object, simply type \( \alpha \alpha \alpha \alpha \alpha \alpha \text{ENTER} \text{RCL} \). You may also type \( \text{ABC} \text{ENTER} \text{RCL} \) or simply \( \text{ABC} \) (that is, press the white menu button below the \( \text{ABC} \) label). Thus, to recall the real number 512 previously stored, press the menu button for \( \text{ABC} \).

If the name \( \text{ABC} \) 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 \( \text{ABC} \).
To create a subdirectory, use the `CREDIR` command, found in the MEMORY menu. You type the name of the intended new directory (for example 'DIREC'), then press `CREDIR`.

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 `EOL` 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...

```

\begin{array}{c}
4: \\
3: \\
2: \\
1: \\
\end{array}
\begin{array}{c}
2 \\
3 \end{array}
```

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

```

\begin{array}{c}
4: \\
3: \\
2: \\
1: \\
\end{array}
\begin{array}{c}
5 \\
\end{array}
```

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

```

\begin{array}{c}
4: \\
3: \\
2: \\
1: \\
\end{array}
\begin{array}{c}
"ABC" \\
"DEF" \\
\end{array}
```

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

```

\begin{array}{c}
4: \\
3: \\
2: \\
1: \\
\end{array}
\begin{array}{c}
"ABCDEF" \\
\end{array}
```

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 \(\text{[«]}\). 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 \(\text{[}\) button allows you to erase the character to the left of the cursor. In the case of a more devastating mistake, pressing \(\text{ATTN}\) (that's the \(\text{ON}\) 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 \[\text{ENTER}\]. The screen should now show:

```
4:
3:
2:
1: \(\leftarrow 3 \wedge\)
```

The program is now on the stack, and it is in level one. We could now execute the program by pressing \[\text{EVAL}\], 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: \(1\alpha\alpha\text{CUBE}1\text{ENTER}\text{STO}\). Now, if you press the button \[\text{VAR}\], you will see \[\text{CUBE1}\] 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 \[\text{CUBE1}\]. The number on the stack will be cubed—with the touch of one 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 executeBYTES. 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 $^\wedge$, 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:

  $$
  4:  
  3:  
  2:  
  1:  
  1: 
  5
  $$

  After the first DUP we would have:

  $$
  4:  
  3:  
  2:  
  1:  
  5  
  5
  $$

  ...after the second DUP:

  $$
  4:  
  3:  
  2:  
  1:  
  5  
  5  
  5
  $$

  ...after the first multiplication:

  $$
  4:  
  3:  
  2:  
  1:  
  5  
  25
  $$

  ...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 \(\ast\), 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 \(\ast\) 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 \(\ast\). A few notes on local variables:

- \(\ast\) 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 \(\ast\) \(\ast\) \(\ast\) \(\ast\) \(\ast\) will place 5 in the variable \(\ast\) and 42 in the variable \(\ast\).
- 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: 
  \[
  \leq 1 \ast \ast 2 \ast \ast \ast \ast \ast \ast
  \]
  1 is placed in the first local variable \(\ast\), then 2 in a local variable of the same name. When \(\ast\) 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 \(\ast\) 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: "\(^{\ast}\) 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: :

\[
\text{CUBE} (\# \text{C875h})
\]

\[
\ast \ast \ast \ast \ast \ast
\]

\[
\text{A}^3
\]

\[
\ast
\]
• 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:

- $f(n) = g(f(n-1))$, where $g$ is a known, calculable function.
- $f(n_0) = f_o$, 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 $f(n_0) = f_o$ is known, then so is $f(f(n_o))$, and $f(f(f(n_o)))$, etc. In other words, to calculate $f(n)$, we use $f(n-1)$ to make the calculation; to calculate $f(n-1)$, we use $f(n-2)$, and so on.

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

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

That is, to calculate $\text{factorial}(n)$, we say:

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

This can be programmed directly:

```plaintext
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 $\text{factorial}(N-1)$ and multiply it by $N$. 

42 Part One: The HP 48
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:

<table>
<thead>
<tr>
<th>Copy 1</th>
<th>Copy 2</th>
<th>Copy 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>It now waits for a response.... N is still 2.</td>
<td>Waiting here, too; N is still 1.</td>
<td>Factorial begins with N = 0. But factorial(0)=1, so the value of 1 is returned immediately to the calling program.</td>
</tr>
<tr>
<td>Still waiting; N is still 2.</td>
<td>Still waiting; N is still 2.</td>
<td>The value of factorial(0) arrives and is multiplied by N to get 1.</td>
</tr>
<tr>
<td>Finally, the value of factorial(1) arrives and is multiplied by N to get 2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Copy 1</th>
<th>Copy 2</th>
<th>Copy 3</th>
<th>Copy 4</th>
<th>Copy 5</th>
<th>Copy 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N=5, f(4)=? )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( N=5\ldots(\text{wait}) )</td>
<td>( N=4, f(3)=? )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( N=5\ldots(\text{wait}) )</td>
<td>( N=4\ldots(\text{wait}) )</td>
<td>( N=3, f(2)=? )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( N=5\ldots(\text{wait}) )</td>
<td>( N=4\ldots(\text{wait}) )</td>
<td>( N=3\ldots(\text{wait}) )</td>
<td>( N=2, f(1)=? )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( N=5\ldots(\text{wait}) )</td>
<td>( N=4\ldots(\text{wait}) )</td>
<td>( N=3\ldots(\text{wait}) )</td>
<td>( N=2\ldots(\text{wait}) )</td>
<td>( N=1, f(0)=? )</td>
<td></td>
</tr>
<tr>
<td>( N=5\ldots(\text{wait}) )</td>
<td>( N=4\ldots(\text{wait}) )</td>
<td>( N=3\ldots(\text{wait}) )</td>
<td>( N=2\ldots(\text{wait}) )</td>
<td>( N=1\ldots(\text{wait}) )</td>
<td>( N=0, f(0)=1 )</td>
</tr>
<tr>
<td>( N=5\ldots(\text{wait}) )</td>
<td>( N=4\ldots(\text{wait}) )</td>
<td>( N=3\ldots(\text{wait}) )</td>
<td>( N=2\ldots(\text{wait}) )</td>
<td>( N=1, f(0)=1 )</td>
<td>( \rightarrow f(1)=1 )</td>
</tr>
<tr>
<td>( N=5\ldots(\text{wait}) )</td>
<td>( N=4\ldots(\text{wait}) )</td>
<td>( N=3\ldots(\text{wait}) )</td>
<td>( N=2, f(1)=1 )</td>
<td>( \rightarrow f(2)=2 )</td>
<td></td>
</tr>
<tr>
<td>( N=5\ldots(\text{wait}) )</td>
<td>( N=4\ldots(\text{wait}) )</td>
<td>( N=3, f(2)=2 )</td>
<td>( \rightarrow f(3)=6 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( N=5\ldots(\text{wait}) )</td>
<td>( N=4, f(3)=6 )</td>
<td>( \rightarrow f(4)=24 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( N=5, f(4)=24 )</td>
<td>( \rightarrow f(5)=120 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 different 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)
{ { "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 pressUSR, 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 pressingUSR (this sequence puts us in 1USR mode, that is, USER mode for one keypress). To remain in USER mode, typeUSRUSR, and USER will appear at the top of the screen. To return to normal mode, typeUSR 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:

\[
arg1 \ arg2 \ \text{ASN}
\]

\(arg1\) 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 \(\rightarrow\) mode (orange shift)
  - 3 \(\rightarrow\) mode (blue shift)
  - 4 \(\alpha\) mode (alpha)
  - 5 \(\alpha\) \(\rightarrow\) mode (alpha, orange shift)
  - 6 \(\alpha\) \(\rightarrow\) mode (alpha, blue shift)

For example, to redefine the \(\text{DROP}\) 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 pre-defined name, 'SKEY'. Or, executing \(\text{DELKEYS}\) 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 « (© ( © © 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 difficulties.
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: \( \star x \colon name \) where \( x \) is the number of the port (0, 1 or 2). For example, if the card is in port 2, then "hello"\( \star 2 \colon \text{BONJOUR} \) \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 is not affected by a 'memory lost'.

- 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

6. Saving and Transmitting Data
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:

<table>
<thead>
<tr>
<th>I/O setup menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR/wire: IR</td>
</tr>
<tr>
<td>ASCII/binary: ascii</td>
</tr>
<tr>
<td>baud: 9600</td>
</tr>
<tr>
<td>parity: none 0</td>
</tr>
<tr>
<td>checksum type: 3</td>
</tr>
<tr>
<td>translate code: 1</td>
</tr>
</tbody>
</table>

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.
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' \( \Rightarrow \partial \)
  Thus, 'SIN(X)/X' 'X' \( \Rightarrow \partial \) 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 (\( \Rightarrow \text{ALGEBRA} \)).

- 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 (\( \Rightarrow \text{SOLVE} \)).
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 different bases). There are also many statistical functions that are available in the STAT 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 \( \underline{2} \)), followed by the characters representing the desired unit.

For example, to create 1_m, you would press 1 \( \underline{2} \) \( \underline{0} \) \( \underline{2} \) \( \underline{0} \) \( \underline{2} \) \( \underline{M} \).

Alternatively, you can place just the value on the stack, then go to the UNITS menu (\( \underline{F} \) \( \underline{U} \) \( \underline{N} \) \( \underline{T} \) \( \underline{S} \)), 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).

\( \underline{F} \) \( \underline{U} \) \( \underline{N} \) \( \underline{T} \) \( \underline{S} \) gives you another UNIT menu with various functions including the CONVERT function which allows conversion between different units.

Time

The TIME menu (\( \underline{F} \) \( \underline{T} \) \( \underline{M} \) \( \underline{E} \)) 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 \( \underline{F} \) \( \underline{T} \) \( \underline{M} \) \( \underline{E} \) 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:

<table>
<thead>
<tr>
<th>address_1</th>
<th>contents_1</th>
<th>length_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>address_2</td>
<td>contents_2</td>
<td>length_2</td>
</tr>
<tr>
<td>address_3</td>
<td>contents_3</td>
<td>length_3</td>
</tr>
<tr>
<td>address_last</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 @, 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 different 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:

```assembly
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).
9. The Saturn Microprocessor

OUTPUT / INPUT masks for the keyboard

A
B
C
D
E
F

MTH
PRG
CST
VAR
▲
NXT

STO
EVAL

SIN
COS
TAN
√x
yx
1/x

ENTER
+/-
EEEX
DEL
←

α
7
8
9
÷

4
5
8
×

1
2
3
−

ON
0
.
SPC
+

400 / 8000
001 / 0008
001 / 0004
001 / 0002
001 / 0001
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:

<table>
<thead>
<tr>
<th>register's nibble number</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>W</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</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 - Byte:** Two nibbles equal one byte;
- **M - Mantissa:** 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 - Sign:** Corresponds to the sign field of the HP 71;
- **X - eXponent:** Corresponds to the exponent field of the HP 71;
- **XS - eXponent Sign:** Corresponds to the HP 71 exponent sign field;
- **W - Wide:** 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: \( Ab=B \ b\), for \( A=0 \ W \), you would use the code \( AF0 \) (\( F \) for \( W \) since the letter given is \( b \)).

Another way to 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 \( P \).

<table>
<thead>
<tr>
<th>Field</th>
<th>( a )</th>
<th>( f )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>( WP )</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>( XS )</td>
<td>2</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>( X )</td>
<td>3</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>( S )</td>
<td>4</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>( M )</td>
<td>5</td>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>( B )</td>
<td>6</td>
<td>6</td>
<td>E</td>
</tr>
<tr>
<td>( W )</td>
<td>7</td>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>( A )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Part Two: Machine Language**
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 different 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?

\[
\begin{align*}
\text{Ba3} & \quad D=D-C \quad a \\
\text{AbB} & \quad C=D \quad \quad b
\end{align*}
\]

9-2. How would you code the above using fields \text{P} and \text{WP}.

9-3. Knowing that:

\[
\begin{align*}
\text{Ra3} & \quad D=D+C \quad a \\
\text{Ab3} & \quad D=0 \quad \quad b
\end{align*}
\]

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=DAT8 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 \textit{instruction field (cycles) code}, where:

- \textit{instruction} is the mnemonic for a particular instruction (e.g.: \texttt{R=0});
- \textit{field} is the field in which the instruction has effect;
- \textit{code} is the hexadecimal code of the instruction.

- \textit{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, \texttt{#00000h} and \texttt{#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 \( n.5 \) in-
struction 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, $g$ 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 DO=(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).

- $DO=(5) 00000 : 10$ 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:
    \[ A = 0 \quad (8) \]
  - Set any other field to zero:
    \[ A = 0 \quad b \quad (4.5+q) \quad Ab0 \]
  - 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:
    \[ ABIT = 0 \quad x \quad (7.5) \quad 8084x \]
  - Set bit \( x \) to one. This is the inverse of the previous instruction.
    \[ ABIT = 1 \quad x \quad (7.5) \quad 8085x \]
  - Move a value into A. This instruction moves \( x+1 \) nibbles into the register (nibbles \( h_0...h_x \)), using the value of P: Nibble \( h_0 \) is moved into nibble \( P \) of A; \( h_j \) is moved into nibble \( P+j \), etc. Remember that the processor reverses the nibbles moved.
    \[ LAHEX(x) \quad h_x...h_0 \quad (5+q+(5+q)/2) \quad 8082xh_0...h_x \]

- **For register B:**
  - Set field A to zero:
    \[ B = 0 \quad A \quad (8) \quad D1 \]
  - Set any other field to zero:
    \[ B = 0 \quad b \quad (4.5+q) \quad Ab1 \]

- **For register C:**
  - Set field A to zero:
    \[ C = 0 \quad A \quad (8) \quad D2 \]
  - Set any other field to zero:
    \[ C = 0 \quad b \quad (4.5+q) \quad Ab2 \]
  - Clear bit \( x \) (0h \( \leq x \) \( \leq Fh \)):
    \[ CBIT = 0 \quad x \quad (7.5) \quad 8088x \]
  - Set bit \( x \) (0h \( \leq x \) \( \leq Fh \)):
    \[ CBIT = 1 \quad x \quad (7.5) \quad 8089x \]
  - Move a value into C:
    \[ LCHEX \quad #h_x...h_0 \quad (2+q+(2+q)/2) \quad 3xh_0...h_x \]
• For register D:
  - Set field A to zero:
    \[ D=0 \quad A \quad (8) \quad D3 \]
  - Set any other field to zero:
    \[ D=0 \quad b \quad (4.5+q) \quad A b 3 \]
• For register P:
  - Move the value \( n \) (0h ≤ n ≤ Fh) into P:
    \[ P = n \quad (3) \quad 2n \]
• For register DO:
  - Move a value into the 2 least significant nibbles:
    \[ D0= (2) \quad qp \quad (6) \quad 19pq \]
  - Move a value into the 4 least significant nibbles:
    \[ D0= (4) \quad srqp \quad (9) \quad 1Apqrs \]
  - Move a value into DO:
    \[ D0= (5) \quad tsrqp \quad (10.5) \quad 1Bpqrst \]
• For register D1:
  - Move a value into the 2 least significant nibbles:
    \[ D1= (2) \quad qp \quad (6) \quad 1Dpq \]
  - Move a value into the 4 least significant nibbles:
    \[ D1= (4) \quad srqp \quad (9) \quad 1Epqrs \]
  - Move a value into D1:
    \[ D1= (5) \quad tsrqp \quad (10.5) \quad 1Fpqrst \]
• For register HST:
  - Clear flag XM:
    \[ XM= 0 \quad (4.5) \quad 821 \]
  - Clear flag SB:
    \[ SB= 0 \quad (4.5) \quad 822 \]
  - Clear flag SR:
    \[ SR= 0 \quad (4.5) \quad 824 \]
  - Clear flag MP:
    \[ MP= 0 \quad (4.5) \quad 828 \]
  - Clear all four flags:
    CLRHST \quad (4.5) \quad 82F \]
• For register ST:
  - Clear flag d (0h ≤ d ≤ Fh):
    \[ ST=0 \quad d \quad (5.5) \quad 84d \]
  - Clear all flags:
    CLRST \quad (7) \quad 08 \]
  - Set flag d:
    \[ ST=1 \quad d \quad (5.5) \quad 85d \]

10. The Saturn Instruction Set
**Moving Values**

- **For Register A:**
  - Move field A of B into field A:
    \[ A = B \]
    \[ A \quad (8) \]
    \[ D4 \]
  - Move field b of B into field b:
    \[ A = B \]
    \[ b \quad (4.5+q) \]
    \[ Ab4 \]
  - The same instructions exist for C:
    \[ A = C \]
    \[ A \quad (8) \]
    \[ DA \]
    \[ A = C \]
    \[ b \quad (4.5+q) \]
    \[ AbA \]

- **For Register B:**
  - Move field A of A into field A:
    \[ B = A \]
    \[ A \quad (8) \]
    \[ D8 \]
  - Move field b of A into field b:
    \[ B = A \]
    \[ b \quad (4.5+q) \]
    \[ Ab8 \]
  - The same instructions exist for C:
    \[ B = C \]
    \[ A \quad (8) \]
    \[ D5 \]
    \[ B = C \]
    \[ b \quad (4.5+q) \]
    \[ Ab5 \]

- **For Register C:**
  - Move field A of A into field A:
    \[ C = A \]
    \[ A \quad (8) \]
    \[ D6 \]
  - Move field b of A into field b:
    \[ C = A \]
    \[ b \quad (4.5+q) \]
    \[ Ab6 \]
  - The same instructions exist for B:
    \[ C = B \]
    \[ A \quad (8) \]
    \[ D9 \]
    \[ C = B \]
    \[ b \quad (4.5+q) \]
    \[ Ab9 \]
  - The same instructions exist for D:
    \[ C = D \]
    \[ A \quad (8) \]
    \[ DB \]
    \[ C = D \]
    \[ b \quad (4.5+q) \]
    \[ AbB \]
  - Move P into nibble n:
    \[ C = P \]
    \[ n \quad (8) \]
    \[ 80Cn \]
  - 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:
    \[ D = C \]
    \[ A \quad (8) \]
    \[ D7 \]
  - Move field b of C into field b:
    \[ D = C \]
    \[ b \quad (4.5+q) \]
    \[ Ab7 \]
• For Register \textbf{P}:
  - Move nibble \( n \) of \textbf{C} into \textbf{P}:
    \[ P = C \quad n \quad (8) \quad 80Dn \]
• For Register \textbf{DO}:
  - Move field \textbf{A} of \textbf{A} into \textbf{DO}:
    \[ D0 = A \quad (9.5) \quad 130 \]
  - Move nibbles 0 to 3 of \textbf{A} into \textbf{DO}:
    \[ D0 = AS \quad (8.5) \quad 138 \]
  - The same instructions exist for \textbf{C}:
    \[ D0 = C \quad (9.5) \quad 134 \]
    \[ D0 = CS \quad (8.5) \quad 13C \]
• For Register \textbf{D1}:
  - Move field \textbf{A} of \textbf{A} into \textbf{D1}:
    \[ D1 = A \quad (9.5) \quad 131 \]
  - Move nibbles 0 to 3 of \textbf{A} into \textbf{D1}:
    \[ D1 = AS \quad (8.5) \quad 139 \]
  - The same instructions exist for \textbf{C}:
    \[ D1 = C \quad (9.5) \quad 135 \]
    \[ D1 = CS \quad (8.5) \quad 13D \]
• For Register \textbf{ST}:
  - Move field \textbf{X} of \textbf{C} into flags 0 to 11 of \textbf{ST}:
    \[ ST = C \quad (7) \quad 0A \]
Saving and Restoring (Rn and RSTK)

- For Register A:
  - Save the entire register:
    
    | Register | Value | Offset |
    |----------|-------|--------|
    | R0       | A     | (20.5) |
    | R1       | A     | (20.5) |
    | R2       | A     | (20.5) |
    | R3       | A     | (20.5) |
    | R4       | A     | (20.5) |
    
    - Save field A only:
    
    | Register | Value | Offset |
    |----------|-------|--------|
    | R0       | A     | (14)   |
    | R1       | A     | (14)   |
    | R2       | A     | (14)   |
    | R3       | A     | (14)   |
    | R4       | A     | (14)   |
    
    - Save field a only:
    
    | Register | Value | Offset |
    |----------|-------|--------|
    | R0       | a     | (9+q)  |
    | R1       | a     | (9+q)  |
    | R2       | a     | (9+q)  |
    | R3       | a     | (9+q)  |
    | R4       | a     | (9+q)  |
    
  - Restore the entire register:
    
    | Register | Value | Offset |
    |----------|-------|--------|
    | A        | R0    | (20.5) |
    | A        | R1    | (20.5) |
    | A        | R2    | (20.5) |
    | A        | R3    | (20.5) |
    | A        | R4    | (20.5) |
    
    - Restore field A only:
    
    | Register | Value | Offset |
    |----------|-------|--------|
    | A        | R0    | (14)   |
    | A        | R1    | (14)   |
    | A        | R2    | (14)   |
    | A        | R3    | (14)   |
    | A        | R4    | (14)   |
    
    - Restore field a only:
    
    | Register | Value | Offset |
    |----------|-------|--------|
    | A        | R0    | (9+q)  |
    | A        | R1    | (9+q)  |
    | A        | R2    | (9+q)  |
    | A        | R3    | (9+q)  |
    | A        | R4    | (9+q)  |

Part Two: Machine Language
• 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:
  \[ \begin{align*}
  \text{C=DAT1} & \quad A \quad (20.5, 3.5) \quad 147 \\
  \text{C=DAT1} & \quad B \quad (19.5) \quad 14F \\
  \text{C=DAT1} & \quad a \quad (20 + q, (q + 2)/2) \quad 157a \\
  \text{C=DAT1} & \quad x+1 \quad (19 + q, (q + 2)/2) \quad 15Fx \\
  \end{align*} \]

- Move field A into the address pointed to by D0:
  \[ \begin{align*}
  \text{DAT0}=\text{C} & \quad A \quad (19.5) \quad 144 \\
  \end{align*} \]

- Same for field B:
  \[ \begin{align*}
  \text{DAT0}=\text{C} & \quad B \quad (16.5) \quad 14C \\
  \end{align*} \]

- Same for field a:
  \[ \begin{align*}
  \text{DAT0}=\text{C} & \quad a \quad (19 + q) \quad 154a \\
  \end{align*} \]

- Same for \( x+1 \) nibbles:
  \[ \begin{align*}
  \text{DAT0}=\text{C} & \quad x+1 \quad (18 + q) \quad 15Cx \\
  \end{align*} \]

- The same instructions exist for D1:
  \[ \begin{align*}
  \text{DAT1}=\text{C} & \quad A \quad (19.5) \quad 145 \\
  \text{DAT1}=\text{C} & \quad B \quad (16.5) \quad 14D \\
  \text{DAT1}=\text{C} & \quad a \quad (19 + q) \quad 155a \\
  \text{DAT1}=\text{C} & \quad x+1 \quad (18 + q) \quad 15Dx \\
  \end{align*} \]

**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 \( \text{A=IN} \) and \( \text{C=IN} \) 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):
    \[ \text{A=IN} \quad (8.5) \quad 802 \]

- For Register C:
  - Read the Input (into nibbles 0,1,2 and 3 of C):
    \[ \text{C=IN} \quad (8.5) \quad 803 \]
  - Write field X to the output:
    \[ \text{OUT=C} \quad (7.5) \quad 801 \]
  - Write nibble 0 into nibble 0 of the output register:
    \[ \text{OUT=CS} \quad (5.5) \quad 800 \]
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)  80Fn

- 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 \( \leq x \leq Fh \)):
    \[ A = A + x + 1 \]  
    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 \( \leq x \leq Fh \)):
    \[ B = B + x + 1 \]  
    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 \( \leq x \leq Fh \)):
    \[ C = C + x + 1 \]  
    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 \( \leq x \leq Fh \)):
  \[ D = D + x + 1 \]  
  \( A \) (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 \( x+1 \):
    \[ D0 = D0 + x \]  
    (8.5) 16x

- For register D1:
  - Increment by \( x+1 \):
    \[ D1 = D1 + x \]  
    (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:
  \[ C = C + B \quad A \quad (8) \quad C9 \]
  \[ C = C + B \quad a \quad (4.5+q) \quad Ra9 \]
- The same instructions exist for D:
  \[ C = C + D \quad A \quad (8) \quad CB \]
  \[ C = C + D \quad a \quad (4.5+q) \quad RaB \]
- Add P+1 to field A:
  \[ C + P + 1 \quad (9.5) \quad 809 \]
  
  • For register D:
    - Add field A of C to field A:
      \[ D = D + C \quad A \quad (8) \quad C3 \]
    - Add field a of C to field a:
      \[ D = D + C \quad a \quad (4.5+q) \quad Ra3 \]

**Decrement**

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

  • For register A:
    - Decrement field A:
      \[ A = A - 1 \quad A \quad (8) \quad CC \]
    - Decrement field a:
      \[ A = A - 1 \quad a \quad (4.5+q) \quad RaC \]
    - Decrement field A by \( x+1 \) (0h ≤ \( x \) ≤ Fh):
      \[ A = A - (x+1) \quad A \quad (13) \quad 818F8x \]
    - Decrement field a by \( x+1 \):*
      \[ A = A - (x+1) \quad a \quad (8+q) \quad 818a8x \]

  • For register B:
    - Decrement field A:
      \[ B = B - 1 \quad A \quad (8) \quad CD \]
    - Decrement field a:
      \[ B = B - 1 \quad a \quad (4.5+q) \quad RaD \]
    - Decrement field A by \( x+1 \) (0h ≤ \( x \) ≤ Fh):
      \[ B = B - (x+1) \quad A \quad (13) \quad 818F9x \]
    - Decrement field a by \( x+1 \):*
      \[ B = B - (x+1) \quad a \quad (8+q) \quad 818a9x \]

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

10. *The Saturn Instruction Set*
• For register C:
  - Decrement field A:
    \[ C = C - 1 \quad A \]
    \( (8) \) \( CE \)
  - Decrement field a:
    \[ C = C - 1 \quad a \]
    \( (4.5+q) \) \( AaE \)
  - Decrement field A by \( x+1 \) (\( 0h \leq x \leq Fh \)):
    \[ C = C - (x+1) \quad A \]
    \( (13) \) \( 818FAx \)
  - Decrement field a by \( x+1 \):
    \[ C = C - (x+1) \quad a \]
    \( (8+q) \) \( 818aAx \)

• For register D:
  - Decrement field A:
    \[ D = D - 1 \quad A \]
    \( (8) \) \( CF \)
  - Decrement field a:
    \[ D = D - 1 \quad a \]
    \( (4.5+q) \) \( AaF \)
  - Decrement field A by \( x+1 \) (\( 0h \leq x \leq Fh \)):
    \[ D = D - (x+1) \quad A \]
    \( (13) \) \( 818FBx \)
  - Decrement field a by \( x+1 \):
    \[ D = D - (x+1) \quad a \]
    \( (8+q) \) \( 818aBx \)

• For register P:
  - Decrement:
    \[ P = P - 1 \]
    \( (4) \) \( OD \)

• For register D0:
  - Decrement by \( x+1 \):
    \[ D0 = D0 - \quad x+1 \]
    \( (8.5) \) \( 18x \)

• For register D1:
  - Decrement by \( x+1 \):
    \[ D1 = D1 - \quad x+1 \]
    \( (8.5) \) \( 1Cx \)

**Subtraction**

These instructions modify the value of the CARRY flag.

• For register A:
  - Subtract field A of C from field A:
    \[ A = A - C \quad A \]
    \( (8) \) \( EA \)
  - Subtract field a of C from field a:
    \[ A = A - C \quad a \]
    \( (4.5+q) \) \( BaA \)

*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:
  \( A = B - A \)  
  \( \text{R} \quad (8) \quad \text{EC} \)

- Subtract field a from field a of B storing the result in field a:
  \( A = B - a \)  
  \( \text{a} \quad (4.5+q) \quad \text{BaC} \)

- For register B:
  - Subtract field A of A from field A:
    \( B = B - A \)  
    \( \text{R} \quad (8) \quad E8 \)
  - Subtract field a of A from field a:
    \( B = B - a \)  
    \( \text{a} \quad (4.5+q) \quad \text{Ba8} \)
  - These same instructions exist for C:
    \( B = B - C \)  
    \( \text{R} \quad (8) \quad E1 \)
    \( B = B - C \)  
    \( \text{a} \quad (4.5+q) \quad \text{Ba1} \)

- Subtract field A from field A of C storing the result in field A:
  \( B = C - B \)  
  \( \text{R} \quad (8) \quad ED \)

- Subtract field a from field a of C storing the result in field a:
  \( B = C - B \)  
  \( \text{a} \quad (4.5+q) \quad \text{BaD} \)

- For register C:
  - Subtract field A of A from field A:
    \( C = C - A \)  
    \( \text{R} \quad (8) \quad E2 \)
  - Subtract field a of A from field a:
    \( C = C - a \)  
    \( \text{a} \quad (4.5+q) \quad \text{Ba2} \)
  - These same instructions exist for D:
    \( C = C - D \)  
    \( \text{R} \quad (8) \quad EB \)
    \( C = C - D \)  
    \( \text{a} \quad (4.5+q) \quad \text{BaB} \)

- Subtract field A from field A of A storing the result in field A:
  \( C = A - C \)  
  \( \text{R} \quad (8) \quad EE \)

- Subtract field a from field a of A storing the result in field a:
  \( C = A - C \)  
  \( \text{a} \quad (4.5+q) \quad \text{BaE} \)

- For register D:
  - Subtract field A of C from field A:
    \( D = D - C \)  
    \( \text{R} \quad (8) \quad E3 \)
  - Subtract field a of C from field a:
    \( D = D - C \)  
    \( \text{a} \quad (4.5+q) \quad \text{Ba3} \)

- Subtract field A from field A of C storing the result in field A:
  \( D = C - D \)  
  \( \text{R} \quad (8) \quad EF \)

- Subtract field a from field a of C storing the result in field a:
  \( D = C - D \)  
  \( \text{a} \quad (4.5+q) \quad \text{BaF} \)
Logical AND

- For register A:
  - Between field A and field A of B:
    \[ A = A \& B \] \( \text{(11) 0EF0} \)
  - Between field a and field a of B:
    \[ A = A \& B \] \( a \text{ (6+q) 0Ea0} \)
  - The same instructions exist for C:
    \[ A = A \& C \] \( \text{(11) 0EF6} \)
    \[ A = A \& C \] \( a \text{ (6+q) 0Ea6} \)

- For register B:
  - Between field A and field A of A:
    \[ B = B \& A \] \( \text{(11) 0EF4} \)
  - Between field a and field a of A:
    \[ B = B \& A \] \( a \text{ (6+q) 0Ea4} \)
  - The same instructions exist for C:
    \[ B = B \& C \] \( \text{(11) 0EF1} \)
    \[ B = B \& C \] \( a \text{ (6+q) 0Ea1} \)

- For register C:
  - Between field A and field A of A:
    \[ C = C \& A \] \( \text{(11) 0EF2} \)
  - Between field a and field a of A:
    \[ C = C \& A \] \( a \text{ (6+q) 0Ea2} \)
  - The same instructions exist for B:
    \[ C = C \& B \] \( \text{(11) 0EF5} \)
    \[ C = C \& B \] \( a \text{ (6+q) 0Ea5} \)
  - The same instructions exist for D:
    \[ C = C \& D \] \( \text{(11) 0EF7} \)
    \[ C = C \& D \] \( a \text{ (6+q) 0Ea7} \)

- For register D:
  - Between field A and field A of C:
    \[ D = D \& C \] \( \text{(11) 0EF3} \)
  - Between field a and field a of C:
    \[ D = D \& C \] \( a \text{ (6+q) 0Ea3} \)
Logical OR

- For register A:
  - Between field A and field A of B:
    \[ A = A \oplus B \quad A \] (11) 0EF8
  - Between field a and field a of B:
    \[ A = A \oplus B \quad a \] (6+q) 0Ea8
  - The same instructions exist for C:
    \[ A = A \oplus C \quad A \] (11) 0EFE
    \[ A = A \oplus C \quad a \] (6+q) 0EaE

- For register B:
  - Between field A and field A of A:
    \[ B = B \oplus A \quad A \] (11) 0EFC
  - Between field a and field a of A:
    \[ B = B \oplus A \quad a \] (6+q) 0EaC
  - The same instructions exist for C:
    \[ B = B \oplus C \quad A \] (11) 0EF9
    \[ B = B \oplus C \quad a \] (6+q) 0Ea9

- For register C:
  - Between field A and field A of A:
    \[ C = C \oplus A \quad A \] (11) 0EFA
  - Between field a and field a of A:
    \[ C = C \oplus A \quad a \] (6+q) 0EaA
  - The same instructions exist for B:
    \[ C = C \oplus B \quad A \] (11) 0EFD
    \[ C = C \oplus B \quad a \] (6+q) 0EaD
  - The same instructions exist for D:
    \[ C = C \oplus D \quad A \] (11) 0EFF
    \[ C = C \oplus D \quad a \] (6+q) 0EaF

- For register D:
  - Between field A and field A of C:
    \[ D = D \oplus C \quad A \] (11) 0EFB
  - Between field a and field a of C:
    \[ D = D \oplus C \quad a \] (6+q) 0EaB

10. The Saturn Instruction Set
**Logical NOT**

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

- For register **A**:
  - On field **A**:
    \[ A = -A - 1 \]
  - On field **b**:
    \[ A = -A - 1 \]
- For register **B**:
  - On field **A**:
    \[ B = -B - 1 \]
  - On field **b**:
    \[ B = -B - 1 \]
- For register **C**:
  - On field **A**:
    \[ C = -C - 1 \]
  - On field **b**:
    \[ C = -C - 1 \]
- For register **D**:
  - On field **A**:
    \[ D = -D - 1 \]
  - On field **b**:
    \[ D = -D - 1 \]

**2's Complement**

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

- For register **A**:
  - On field **A**:
    \[ A = -A \]
  - On field **b**:
    \[ A = -A \]
- For register **B**:
  - On field **A**:
    \[ B = -B \]
  - On field **b**:
    \[ B = -B \]
• For register C:
  - On field A:
    \[ C = -C \]
    \[ A \quad (8) \quad FA \]
  - On field b:
    \[ C = -C \]
    \[ b \quad (4.5+q) \quad BbA \]

• For register D:
  - On field A:
    \[ D = -D \]
    \[ A \quad (8) \quad FB \]
  - On field b:
    \[ D = -D \]
    \[ b \quad (4.5+q) \quad BbB \]

**Multiplying by 2**

• For register A:
  - Multiply field A by 2:
    \[ A = A + A \]
    \[ A \quad (8) \quad C4 \]
  - Multiply field a by 2:
    \[ A = A + A \]
    \[ a \quad (4.5+q) \quad Aa4 \]

• For register B:
  - Multiply field A by 2:
    \[ B = B + B \]
    \[ A \quad (8) \quad C5 \]
  - Multiply field a by 2:
    \[ B = B + B \]
    \[ a \quad (4.5+q) \quad Aa5 \]

• For register C:
  - Multiply field A by 2:
    \[ C = C + C \]
    \[ A \quad (8) \quad C6 \]
  - Multiply field a by 2:
    \[ C = C + C \]
    \[ a \quad (4.5+q) \quad Aa6 \]

• For register D:
  - Multiply field A by 2:
    \[ D = D + D \]
    \[ A \quad (8) \quad C7 \]
  - Multiply field a by 2:
    \[ D = D + D \]
    \[ a \quad (4.5+q) \quad Aa7 \]
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
  - 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
  - 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
  - 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
  - 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:
  \( \text{ASLC} \) (22.5) 810

• For register B:
  \( \text{BSLC} \) (22.5) 811

• For register C:
  \( \text{CSLC} \) (22.5) 812

• For register D:
  \( \text{DSLC} \) (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 6aaa, where aaa is the jump distance. Thus, to jump between addresses $@_1$ and $@_2$:

- If the jump is forward, $(@_2 - (@_1 + I))$ calculates the distance. You add 1 to $@_1$ because that’s the length of the jump instruction 6aaa (you don’t count the nibbles 6aaa 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, $((@_1 + I) - @_2)$ 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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Length</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTO abc</td>
<td></td>
<td>(14)</td>
<td>6cba</td>
</tr>
<tr>
<td>GOLONG abcd</td>
<td></td>
<td>(17)</td>
<td>8Cdcba</td>
</tr>
</tbody>
</table>

**Direct relative conditional**

These jumps depend on the state of the CARRY flag.

- Jump on CARRY clear:
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Length</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>GONC ab</td>
<td></td>
<td>(12.5/4.5)</td>
<td>5ba</td>
</tr>
</tbody>
</table>

- Jump on CARRY set:
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Length</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOC ab</td>
<td></td>
<td>(12.5/4.5)</td>
<td>4ba</td>
</tr>
</tbody>
</table>
### Absolute

\[
\text{GOVLNG abcde (18.5) 80dedcba}
\]

### Register direct

- **Using register A:**
  - Jump to the address contained in field A:
    \[
    PC = A \quad (19) \quad 81B2
    \]
  - Jump to the address contained in field A, saving the address of the next instruction into field A:
    \[
    APCex \quad (19) \quad 81B6
    \]
- **Using register C:**
  - Jump to the address contained in field A:
    \[
    PC = C \quad (19) \quad 81B3
    \]
  - Jump to the address contained in field A, saving the address of the next instruction into field A:
    \[
    CPCex \quad (19) \quad 81B7
    \]

### 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):
    \[
    PC = (A) \quad (26, 3.5) \quad 808C
    \]
- **Using register C:**
  - Jump to the address contained in the 5 nibbles pointed to by field C:
    \[
    PC = (C) \quad (26, 3.5) \quad 808E
    \]

### 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:**
  \[
  A = PC \quad (11) \quad 81B4
  \]
- **Move PC into field A of register C:**
  \[
  C = PC \quad (11) \quad 81B5
  \]
Calling Subroutines

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

\[
\text{GOSUB } @_1 \\
@_2 \quad \text{(next instruction)} \\
@_1 \quad \text{(some useful subroutine)}
\]

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**

- \( \text{GOSUB} \ abc \ (15) \ 7\text{bca} \)
- \( \text{GOSUBL} \ abcd \ (18) \ 8\text{Edcba} \)

**Absolute**

- \( \text{GOSBVL} \ abcde \ (19.5) \ 8\text{Fedcba} \)

**Returning From Subroutines**

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

All comparisons are of the form

\[ ? <\text{register}> <\text{comparator}> <\text{register or immediate}> <\text{field}> \]

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?  
    \[ ?A=0 \quad A \quad (21.5/13.5) \quad 8A8 \]
  - Is field a zero?  
    \[ ?A=0 \quad a \quad (16.5+q/8.5+q) \quad 9a8 \]
  - Is field A non zero?  
    \[ ?A\#0 \quad A \quad (21.5/13.5) \quad 8AC \]
  - Is field a non zero?  
    \[ ?A\#0 \quad a \quad (16.5+q/8.5+q) \quad 9aC \]
  - Is bit x (0h \leq x \leq Fh) clear?  
    \[ ?\text{ABIT}=0 \quad x \quad (20.5/12.5) \quad 8086x \]
  - Is bit x (0h \leq x \leq Fh) set?  
    \[ ?\text{ABIT}=1 \quad x \quad (20.5/12.5) \quad 8087x \]
• For register B:
  - Is field A zero?
    \( ?B=0 \) \( \text{A} \) (21.5/13.5) 8A9
  - Is field a zero?
    \( ?B=0 \) \( \text{a} \) (16.5+q/8.5+q) 9a9
  - Is field A non zero?
    \( ?B\neq0 \) \( \text{A} \) (21.5/13.5) 8AD
  - Is field a non zero?
    \( ?B\neq0 \) \( \text{a} \) (16.5+q/8.5+q) 9aD

• For register C:
  - Is field A zero?
    \( ?C=0 \) \( \text{A} \) (21.5/13.5) 8AA
  - Is field a zero?
    \( ?C=0 \) \( \text{a} \) (16.5+q/8.5+q) 9aA
  - Is field A non zero?
    \( ?C\neq0 \) \( \text{A} \) (21.5/13.5) 8AE
  - Is field a non zero?
    \( ?C\neq0 \) \( \text{a} \) (16.5+q/8.5+q) 9aE
  - Is bit \( x \) (0h ≤ \( x \) ≤ Fh) clear?
    \( ?CBIT=0 \) \( \text{x} \) (20.5/12.5) 808Ax
  - Is bit \( x \) (0h ≤ \( x \) ≤ Fh) set?
    \( ?CBIT=1 \) \( \text{x} \) (20.5/12.5) 808Bx

• For register D:
  - Is field A zero?
    \( ?D=0 \) \( \text{A} \) (21.5/13.5) 8AB
  - Is field a zero?
    \( ?D=0 \) \( \text{a} \) (16.5+q/8.5+q) 9aB
  - Is field A non zero?
    \( ?D\neq0 \) \( \text{A} \) (21.5/13.5) 8AF
  - Is field a non zero?
    \( ?D\neq0 \) \( \text{a} \) (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) 89n
- Is P not equal to n?
  \( ?P\# n \) (15.5/7.5) 88n

For register ST:
- Is flag n clear?
  \( ?ST=0 n \) (16.5/8.5) 86n
- Is flag n set?
  \( ?ST=1 n \) (16.5/8.5) 87n
- Is flag n not clear?
  \( ?ST\#0 n \) (16.5/8.5) 87n
- Is flag n not set?
  \( ?ST\#1 n \) (16.5/8.5) 86n

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?
  \(?A\geq B\)  A  (21.5/13.5)  8B8
- Is field a greater than or equal to field a of register B?
  \(?a\geq b\)  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?
    \(?B\neq A\)  A  (21.5/13.5)  8A4
  - Is field a not equal to field a of register A?
    \(?B\neq A\)  a  (16.5+q/8.5+q)  9a4
  - The same instructions exist for C:
    \(?B\neq C\)  A  (21.5/13.5)  8A5
    \(?B\neq C\)  a  (16.5+q/8.5+q)  9a5
  - Is field A less than or equal to field A of register C?
    \(?B\leq C\)  A  (21.5/13.5)  8B0
  - Is field a less than or equal to field a of register C?
    \(?B\leq C\)  b  (16.5+q/8.5+q)  9b0
  - 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?
    \(?B\geq C\)  A  (21.5/13.5)  8B9
  - Is field a greater than or equal to field a of register C?
    \(?B\geq C\)  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

116  

Part Two: Machine Language
• 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

10. The Saturn Instruction Set 117
• 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 A?
    \( ?D#C \) \( a \) \((16.5+q/8.5+q)\) \( 9a7 \)
  - Is field A less than or equal to field A of register C?
    \( ?D\leq C \) \( A \) \((21.5/13.5)\) \( 8BF \)
  - Is field a less than or equal to field a of register A?
    \( ?D\leq C \) \( b \) \((16.5+q/8.5+q)\) \( 9bF \)
  - 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 A?
    \( ?D< C \) \( b \) \((16.5+q/8.5+q)\) \( 9b7 \)
  - Is field A greater than or equal to field A of register C?
    \( ?D\geq C \) \( A \) \((21.5/13.5)\) \( 8BB \)
  - Is field a greater than or equal to field a of register A?
    \( ?D\geq C \) \( b \) \((16.5+q/8.5+q)\) \( 9bB \)
  - 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 A?
    \( ?D> C \) \( b \) \((16.5+q/8.5+q)\) \( 9b3 \)
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:
    \[
    \text{INTON} \quad (7) \quad 8080
    \]
  - Disable maskable interrupts:
    \[
    \text{INTOFF} \quad (7) \quad 808F
    \]
  - Clear all interrupts:
    \[
    \text{RSI} \quad (8.5) \quad 80810
    \]
- These instructions change the calculation mode for mathematical operations as described in Chapter 2:
  - Set mode to decimal:
    \[
    \text{SETDEC} \quad (4) \quad 05
    \]
  - Set mode to hexadecimal:
    \[
    \text{SETHEX} \quad (4) \quad 04
    \]

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:

\[
\begin{align*}
\text{NOP3} & \quad 420 \\
\text{NOP4} & \quad 6300 \\
\text{NOP5} & \quad 64000
\end{align*}
\]

Pseudo Operations

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

\[
\text{CON}(n) \quad q_1 \ldots q_n \quad q_n \ldots q_1
\]
Exercises

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

```
CON(5) #02DCC
begin CON(5) (end)-(begin)
    GOTO 11
sub1 A=A-1 A
    LCHEX #12345
12    C=C-1 A
    GONCC 12
RTNCC
11    LCHEX #00005
    A=C A
13    GOsub 12
?A#0 A
    GOYES 13
    LCHEX #00001
    A=C A
    GOxub 14
?A=0 A
    GOYES 15
    A=A-1 A
15    A=DAT0 A
    D0=D0+ 5
    PC=(A)
14    ?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`):

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

<table>
<thead>
<tr>
<th>@</th>
<th>Prolog (02911)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>Content</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+Ah</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>`.

---

*Part Two: Machine Language*
Real Number Object

<table>
<thead>
<tr>
<th>@</th>
<th>Prolog (02933)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>Exponent</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>@+8h</td>
<td>Mantissa</td>
<td>12 nibbles</td>
</tr>
<tr>
<td>@+14h</td>
<td>Sign</td>
<td>1 nibble</td>
</tr>
<tr>
<td>@+15h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

\[
\text{sign} \times \text{mantissa} \times 10^{\text{exponent}}
\]

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 33920400009876543210.
- -3.14159265359E-2 is coded as 339208999535629514139.

**Exercises**

11-4. Code the real number 12.
11-5. What does 3392040000000543779 represent?
Long Real Number Object

<table>
<thead>
<tr>
<th></th>
<th>Prolog (02955)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>Exponent</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+Ah</td>
<td>Mantissa</td>
<td>15 nibbles</td>
</tr>
<tr>
<td>@+19h</td>
<td>Sign</td>
<td>1 nibble</td>
</tr>
<tr>
<td>@+1Ah</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 559208765400000000003219

**Exercises**

11-6. How would the HP 48 code the long real 12345678901234567?
11-7. What does 5592089999000000000000019 represent?
Complex Number Object

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02977)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+5h</td>
<td>Exponent 1</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>@+8h</td>
<td>Mantissa 1 real part</td>
<td>12 nibbles</td>
</tr>
<tr>
<td>@+14h</td>
<td>Sign 1</td>
<td>1 nibble</td>
</tr>
<tr>
<td>@+15h</td>
<td>Exponent 2 imaginary part</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>@+18h</td>
<td>Mantissa 2</td>
<td>12 nibbles</td>
</tr>
<tr>
<td>@+24h</td>
<td>Sign 2</td>
<td>1 nibble</td>
</tr>
<tr>
<td>@+25h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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?
\(779201000000000003910000000000330\)
Long Complex Number Object

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (0299D)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+5h</td>
<td>Exponent 1, real</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+Ah</td>
<td>Mantissa 1, real part</td>
<td>15 nibbles</td>
</tr>
<tr>
<td>@+19h</td>
<td>Sign 1</td>
<td>1 nibble</td>
</tr>
<tr>
<td>@+1Ah</td>
<td>Exponent 2, imaginary</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+1Fh</td>
<td>Mantissa 2, imaginary part</td>
<td>15 nibbles</td>
</tr>
<tr>
<td>@+2Eh</td>
<td>Sign 2</td>
<td>1 nibble</td>
</tr>
<tr>
<td>@+2Fh</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Prolog (029BF)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Character</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+5h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@+7h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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 \[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\] is coded as:

```
8E920 95000 33920 20000 20000 20000
00000000000001 0 00000000000002 0
00000000000003 0 00000000000004 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

<table>
<thead>
<tr>
<th>@</th>
<th>Prolog (02A0A)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>Total length excluding prolog l₁</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+Ah</td>
<td>Type of objects (of length l₀)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+Fh</td>
<td>Number, d, of dimensions</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+14h</td>
<td>Dimension 1 (d₁)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+d*5+14h</td>
<td>Dimension d (d_d)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+d*5+19h</td>
<td>Pointer to object 1</td>
<td>5 nibbles</td>
</tr>
<tr>
<td></td>
<td>Pointer to object d_2+1</td>
<td>5 nibbles</td>
</tr>
<tr>
<td></td>
<td>Pointer to object d_1*...*d_d</td>
<td>l₀ nibbles</td>
</tr>
<tr>
<td>@+l₁-l₀+5h</td>
<td>Element 1</td>
<td>l₀ nibbles</td>
</tr>
<tr>
<td>@+l₁+5h</td>
<td>Element n</td>
<td>l₀ nibbles</td>
</tr>
</tbody>
</table>

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 000000000000010 000000000000000
  ```
String Object

<table>
<thead>
<tr>
<th>@</th>
<th>Prolog (02A2C)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>Total length excluding prolog</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+Ah</td>
<td>First character</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+l-2h</td>
<td>Last character</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+l+5h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 I₁ | 5 nibbles |
| @+Ah   | Binary integer value | 1₁-5 nibbles |
| @+I₁+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 E4A20510008765432100000000

**Exercises**

11-18. Code the binary integer #87654321d
11-19. Decode E4A20A000012345
## List Object

<table>
<thead>
<tr>
<th>@</th>
<th>Prolog (02A74)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>First object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Last object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Epilog (0312B)</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>

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 47A20C2A2070001484E201024B2130

### Exercises
11-20. Code an empty list.
Directory Object

| @      | Prolog (02A96)          | 5 nibbles |
| @+5h   | Number of attached libraries, n₁ | 3 nibbles |
| @+8h   | N° Library               | 3 nibbles |
| @+Bh   | Address of Hash Table 1 | 5 nibbles |
| @+10h  | Address of Message Table 1 | 5 nibbles |

| Library n₁ | 3 nibbles |
| Address of Hash Table n₁ | 5 nibbles |
| Address of Message Table n₁ | 5 nibbles |
| Offset to last object (@₃-@₁) | 5 nibbles |
| @₁+5h | 00000 | 5 nibbles |
| @₂    | n₁ characters in name₁ | 2 nibbles |
| @₂+2h | Character 1, name₁ name of object 1 | 2 nibbles |
| character n₁ | 2 nibbles |
| n₁ characters in name₁ | 2 nibbles |
| Object 1 | |
| @₃    | Size of previous fields (@₃@₂) | 5 nibbles |
| @₃+2h | n₂ characters in name₂ | 2 nibbles |
| @₅    | n₅ characters in name₅ | 2 nibbles |
| @₅+2h | Character 1, name₅ name of object d | 2 nibbles |
| Character n₅ | 2 nibbles |
| n₅ characters in name₅ | 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:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
<th>Nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02A96)</td>
<td>5</td>
</tr>
<tr>
<td>@+5h</td>
<td>Number of the attached library</td>
<td>3</td>
</tr>
<tr>
<td>@+8h</td>
<td>Offset to last object (@ - @)</td>
<td>5</td>
</tr>
<tr>
<td>@+Dh</td>
<td>00000</td>
<td>5</td>
</tr>
<tr>
<td>@+12h</td>
<td>n, characters in name</td>
<td>2</td>
</tr>
</tbody>
</table>

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': 69A20FF7A000000000104410C2A207000033

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

<table>
<thead>
<tr>
<th>@</th>
<th>Prolog (02AB8)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>First object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Last object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Epilog (0312B)</td>
<td></td>
</tr>
</tbody>
</table>

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:
  8BA2084E20103484E2010476BA1B2130

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

<table>
<thead>
<tr>
<th>@</th>
<th>Prolog (02ADA)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>Object implied</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Desc 1</td>
<td>unit</td>
</tr>
<tr>
<td></td>
<td>Desc n</td>
<td>description</td>
</tr>
<tr>
<td></td>
<td>Epilog (0312B)</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>*</th>
<th>/</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>#10B86h</td>
<td>#10B68h</td>
<td>#10B72h</td>
</tr>
</tbody>
</table>

**Example**

- 9.81_m/s^2 is coded as

  ADA203392000000000000001890C2A2070000D6 ...
  ...C2A2070000373392000000000000002027B01 ...
  ...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: ADA20339200000000000000150C2A2070000D6 ...
       ...F2A227B0168B01B2130
Tagged Object

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02AFC)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+5h</td>
<td>Length ( I_\text{c} ) of the tag</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+7h</td>
<td>Character 1, characters of the tag</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+( I_\text{c} )*2+5h</td>
<td>Character ( I_\text{c} ), object</td>
<td>2 nibbles</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02B1E)</td>
<td>5</td>
</tr>
<tr>
<td>@+5h</td>
<td>Total length excluding prolog l₁</td>
<td>5</td>
</tr>
<tr>
<td>@+Ah</td>
<td>Number n₁ of lines (in pixels)</td>
<td>5</td>
</tr>
<tr>
<td>@+Fh</td>
<td>Number nᵢ of columns (in pixels)</td>
<td>5</td>
</tr>
<tr>
<td>@+14h</td>
<td>Columns 1 to 8 Pixels in line 1</td>
<td>1+1</td>
</tr>
<tr>
<td></td>
<td>Last pixels</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Columns 1 to 8 Pixels in line nᵢ</td>
<td>1+1</td>
</tr>
<tr>
<td></td>
<td>Last pixels</td>
<td>1+1</td>
</tr>
</tbody>
</table>

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: E1B2011000100000000FF

### Exercise
11-30. Decode: E1B2011000100040000F0
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 \( \geq \#700h \)), 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 \( \leq \#6FFh \)), 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 @+1+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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02A4E)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@h+5h</td>
<td>Total length excluding prolog l_h</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@c1</td>
<td>Offset for class 1 (@n1-@c1)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@c16</td>
<td>Offset for class 16 (@n16-@c16)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@n1+5Ah</td>
<td>Length l_n of the name list</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@n1</td>
<td>Number of characters in name</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@n1+2h</td>
<td>First character Characters in name 1</td>
<td>2 nibbles</td>
</tr>
<tr>
<td></td>
<td>Last character</td>
<td>2 nibbles</td>
</tr>
<tr>
<td></td>
<td>Command number 1</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>@nx</td>
<td>Number of characters in name x</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@nx+2h</td>
<td>First character Characters in name x</td>
<td>2 nibbles</td>
</tr>
<tr>
<td></td>
<td>Last character</td>
<td>2 nibbles</td>
</tr>
<tr>
<td></td>
<td>Command number x</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>@+ln+5Ah</td>
<td>Offset to cmd name 1 (@n1-@n1)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@n</td>
<td>Offset to cmd name x (@n1-@nx)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td></td>
<td>Offset to the last command name</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>Prolog</td>
<td>$029E$ bytes</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>$m+5h$</td>
<td>Total length excluding prolog $l_m$</td>
<td></td>
<td>5 nibbles</td>
</tr>
<tr>
<td>$m+Ah$</td>
<td>Object types: string (02A2C)</td>
<td></td>
<td>5 nibbles</td>
</tr>
<tr>
<td>$m+Fh$</td>
<td>Number of dimensions (00001)</td>
<td></td>
<td>5 nibbles</td>
</tr>
<tr>
<td>$m+14h$</td>
<td>$n$ number of messages</td>
<td></td>
<td>5 nibbles</td>
</tr>
<tr>
<td>$m+19h$</td>
<td>Length $l_1$</td>
<td></td>
<td>5 nibbles</td>
</tr>
<tr>
<td>$m+1Dh$</td>
<td>First character</td>
<td>Text for message 1</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>$m+l_1+19h$</td>
<td>Last character</td>
<td></td>
<td>2 nibbles</td>
</tr>
<tr>
<td>$m+l_n+19h$</td>
<td>Length $l_n$</td>
<td></td>
<td>5 nibbles</td>
</tr>
<tr>
<td>$m+l_n+1Dh$</td>
<td>First character</td>
<td>Text for message n</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>$m+l_n+5h$</td>
<td>Last character</td>
<td></td>
<td>2 nibbles</td>
</tr>
</tbody>
</table>

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:

| @_i     | Prolog (02A4E)                          | 5 nibbles |
| @_i+5h  | Total length excluding prolog i          | 5 nibbles |
| @_ii    | Offset to object 1 (@_o1-@_ii)          | 5 nibbles |
| @_id    | Offset to object d (@_od-@_id)          | 5 nibbles |
| @_i+1+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
  04B20C000400659445440FF60000000000000000000049B1

**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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02B62)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+5h</td>
<td>Total length excluding prolog l₁</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+Ah</td>
<td>n₁ number of characters</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+Ch</td>
<td>Character 1</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n₁*2 +8h</td>
<td>Character n₁</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n₁*2 +Ah</td>
<td>n₁ number of characters</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n₁*2 +Ch</td>
<td>First Backup object</td>
<td></td>
</tr>
<tr>
<td>@+l₁+5h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@</td>
<td>Prolog 02911 System Binary containing CRC</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+l₁</td>
<td>0</td>
<td>1 nibbles</td>
</tr>
<tr>
<td>@+l₁+1h</td>
<td>CRC value</td>
<td>4 nibbles</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02B62)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+5h</td>
<td>Total length excluding prolog l₁</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+Ah</td>
<td>n₁ number of characters</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+Ch</td>
<td>Character 1</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n₁*2 +8h</td>
<td>Character n₁</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n₁*2 +Ah</td>
<td>n₁ number of characters</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n₁*2 +Ch</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>@+l₁-5h</td>
<td>Prolog 02911 System Binary containing CRC</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+l₁</td>
<td>0</td>
<td>1 nibbles</td>
</tr>
<tr>
<td>@+l₁+1h</td>
<td>CRC value</td>
<td>4 nibbles</td>
</tr>
</tbody>
</table>

---

148 | **Part Two: Machine Language**
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 @+11h, inclusive.

**Example**

- 26B2892000402434B40549C282090000F4B41192006D26 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?
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

<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td></td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+5h</td>
<td></td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+Ah</td>
<td>Total length excluding prolog</td>
<td>l,-5 nibbles</td>
</tr>
<tr>
<td>@+l,-5h</td>
<td>Contents</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>@</th>
<th>Prolog (02D9D)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>First object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Last object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Epilog (0312B)</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>

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?
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02E48)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+5h</td>
<td>(n_c) number of characters</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+7h</td>
<td>Character 1</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n_c*2+3h</td>
<td>Characters of (n_c) (th) name</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n_c*2+5h</td>
<td>Character (n_c)</td>
<td>2 nibbles</td>
</tr>
</tbody>
</table>

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:
  
  \[84E2070A1F65727E65697\]

**Exercises**

11-40. Code 'Hello'.

11-41. What does 84E2000 represent?
Local Name Object

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02E6D)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>@+5h</td>
<td>number of characters</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+7h</td>
<td>Character 1</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n_c*2+3h</td>
<td>Character n_c</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>@+n_c*2+5h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>@</th>
<th>Prolog (02E92)</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>@+5h</td>
<td>Library number</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>@+8h</td>
<td>Command number</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>@+Bh</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 be 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: 29E202008361

**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 
- Commands for accessing pre-defined memory areas:
  - #00100h (I/O RAM) by ENTER
  - #80000h (Port 1) by EX
  - #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.

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>#00000h</td>
<td>Beginning of ROM</td>
<td>256 nibbles</td>
</tr>
<tr>
<td>#00100h</td>
<td>I/O RAM</td>
<td>64 nibbles</td>
</tr>
<tr>
<td>#00140h</td>
<td>Continuation of ROM</td>
<td>458432 nibbles</td>
</tr>
<tr>
<td>#70000h</td>
<td>Built-in RAM</td>
<td>65536 nibbles</td>
</tr>
<tr>
<td>#80000h</td>
<td>Port 1 Plug-in</td>
<td>262144 nibbles</td>
</tr>
<tr>
<td>#C0000h</td>
<td>Port 2 cards</td>
<td>262144 nibbles</td>
</tr>
<tr>
<td>#100000h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1: HP 48 memory, standard state**

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>#00000h</td>
<td>Beginning of ROM</td>
<td>256 nibbles</td>
</tr>
<tr>
<td>#00100h</td>
<td>I/O RAM</td>
<td>64 nibbles</td>
</tr>
<tr>
<td>#00140h</td>
<td>Continuation of ROM</td>
<td>523968 nibbles</td>
</tr>
<tr>
<td>#80000h</td>
<td>Port 1 Plug-in</td>
<td>262144 nibbles</td>
</tr>
<tr>
<td>#C0000h</td>
<td>Port 2 (partial) cards</td>
<td>196608 nibbles</td>
</tr>
<tr>
<td>#F0000h</td>
<td>Built-in RAM (displaced)</td>
<td>65536 nibbles</td>
</tr>
<tr>
<td>#100000h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#00100h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00101h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>#00100h</td>
<td>Display</td>
<td>Left margin</td>
<td></td>
</tr>
<tr>
<td>#00101h</td>
<td>Screen contrast</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00102h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00103h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00104h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00105h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00106h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00107h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00108h</td>
<td>Batt. test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00109h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0010Ah</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0010Bh</td>
<td>Alert</td>
<td>Alpha</td>
<td>right shift</td>
</tr>
<tr>
<td>#0010Ch</td>
<td>annunciator</td>
<td>transmitting</td>
<td>Busy</td>
</tr>
<tr>
<td>#0010Dh</td>
<td></td>
<td>RS232 speed</td>
<td></td>
</tr>
<tr>
<td>#0010 Eh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0010Fh</td>
<td>Port information (HP 48SX)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00110h</td>
<td>RS 232C interrupts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00111h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00112h</td>
<td></td>
<td>input OK</td>
<td>output OK</td>
</tr>
<tr>
<td>#00113h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00114h</td>
<td>RS 232C Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00115h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00116h</td>
<td>RS 232C Output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00117h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00118h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00119h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0011 Ah</td>
<td>IR input</td>
<td>R in mem</td>
<td></td>
</tr>
<tr>
<td>#0011 Bh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0011 Ch</td>
<td>IR output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0011 Dh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0011 Eh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0011 Fh</td>
<td>Base address of built-in RAM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 microprocessor 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 (a, x, 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:

<table>
<thead>
<tr>
<th>Bit 2</th>
<th>Bit 1</th>
<th>RS-232C Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1200 Baud</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2400 Baud</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4800 Baud</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>9600 Baud</td>
</tr>
</tbody>
</table>

Port Information (HP 48SX)

Nibble #0010Fh gives the states of the two ports for the HP 48SX. The possible states are:

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>When set (1): Card present in port 1</td>
</tr>
<tr>
<td>1</td>
<td>When set (1): Card present in port 2</td>
</tr>
<tr>
<td>2</td>
<td>When set (1): Card in port 1 not write-protected</td>
</tr>
<tr>
<td>3</td>
<td>When set (1): Card in port 2 not write-protected</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>When set (1): a character was received; an interrupt has occurred.</td>
</tr>
<tr>
<td>1</td>
<td>When set (1): receive interrupts are enabled.</td>
</tr>
<tr>
<td>2</td>
<td>When set (1): a character was transmitted; an interrupt has occurred.</td>
</tr>
<tr>
<td>3</td>
<td>When set (1): transmission interrupts are enabled.</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#00120h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00121h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00122h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00123h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00124h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00125h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00126h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00127h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00128h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00129h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0012Ah</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0012Bh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0012Ch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0012Dh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0012 Eh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0012Fh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00130h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00131h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00132h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00133h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00134h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00135h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00136h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00137h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00138h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#00139h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0013Ah</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0013Bh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0013Ch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0013Dh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0013 Eh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0013Fh</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Beginning address of screen bitmap
- Right margin (in nibbles)
- Menu bar height & VSYNC
- Beginning address of menu bar bitmap
- Timer 1
- Timer 2
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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#70000h</td>
<td>Reserved RAM</td>
</tr>
<tr>
<td>(#70551h)</td>
<td>Screen GROBS</td>
</tr>
<tr>
<td>(#7056Ah)</td>
<td>Temporary objects</td>
</tr>
<tr>
<td>(#7056Fh)</td>
<td>Return stack</td>
</tr>
<tr>
<td>B</td>
<td>Free memory</td>
</tr>
<tr>
<td>D1</td>
<td>The stack</td>
</tr>
<tr>
<td>(#7057Eh)</td>
<td>Command line</td>
</tr>
<tr>
<td>(#70583h)</td>
<td>Undo stack, local variables</td>
</tr>
<tr>
<td>(#70588h)</td>
<td>5 zeros</td>
</tr>
<tr>
<td>(#7058Dh)</td>
<td>Temporary environment</td>
</tr>
<tr>
<td>(#70592h)</td>
<td>User variables (HOME dir)</td>
</tr>
<tr>
<td>(#70597h)</td>
<td>Backup in port 0</td>
</tr>
<tr>
<td>(#70669h)</td>
<td>D*5 nibbles</td>
</tr>
<tr>
<td>(#7069Fh)</td>
<td>(#7069Fh) nib.</td>
</tr>
<tr>
<td></td>
<td>48 nibbles min.</td>
</tr>
<tr>
<td></td>
<td>5 nibbles</td>
</tr>
<tr>
<td></td>
<td>78 nibbles</td>
</tr>
</tbody>
</table>

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.
### 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.
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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Cause of Warm Boot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The machine was turned on while in the COMA mode (COMA mode is entered by pressing ON-SPC).</td>
</tr>
<tr>
<td>1</td>
<td>Batteries are very weak.</td>
</tr>
<tr>
<td>2</td>
<td>A hardware problem occurred during an infrared transmission.</td>
</tr>
<tr>
<td>3</td>
<td>The machine experienced a restart (execution of the program at #00000h).</td>
</tr>
<tr>
<td>4</td>
<td>The clock offset (controlled by CRC) was corrupted.</td>
</tr>
<tr>
<td>5</td>
<td>An uncontrolled data change occurred in one of the plug-in cards.</td>
</tr>
<tr>
<td>6</td>
<td>Not used.</td>
</tr>
<tr>
<td>7</td>
<td>A verification word (5 nibbles) in RAM does not correspond to the memory state (RAM is probably corrupted).</td>
</tr>
<tr>
<td>8</td>
<td>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.</td>
</tr>
<tr>
<td>9</td>
<td>The alarm list is corrupted (its CRC is not valid).</td>
</tr>
<tr>
<td>A</td>
<td>Not used.</td>
</tr>
<tr>
<td>B</td>
<td>Plug-in card removed.</td>
</tr>
<tr>
<td>C</td>
<td>System reset (using the reset button found underneath one of the machine's rubber feet).</td>
</tr>
<tr>
<td>D</td>
<td>RPL error manager not found.</td>
</tr>
<tr>
<td>E</td>
<td>Configuration table corrupted.</td>
</tr>
<tr>
<td>F</td>
<td>RAM card removed.</td>
</tr>
</tbody>
</table>

Next is the date of the warm boot (in 8192nds 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 offset (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).
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>#700B8h</td>
<td>???...???</td>
<td>35 nibbles</td>
</tr>
<tr>
<td>#700DBh</td>
<td>Plug-in cards (bits 0 and 1)</td>
<td>1 nibble</td>
</tr>
<tr>
<td>#700DCh</td>
<td>Data</td>
<td>288 nibbles</td>
</tr>
<tr>
<td>#701FCh</td>
<td>BufLen for the Input buffer for the RS 232C port</td>
<td>512 nibbles</td>
</tr>
<tr>
<td>#703FEh</td>
<td>BufFull</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>#703FFh</td>
<td>BufStart</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>#70401h</td>
<td></td>
<td>39 nibbles</td>
</tr>
<tr>
<td>#70428h</td>
<td>CRC for the configuration table</td>
<td>4 nibbles</td>
</tr>
<tr>
<td>#7042Ch</td>
<td>Flags Information for the plug-in</td>
<td>1 nibble</td>
</tr>
<tr>
<td>#7042Dh</td>
<td>Size Information for the plug-in</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70432h</td>
<td>Start card in port 1</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70437h</td>
<td>Flags Information for the plug-in</td>
<td>1 nibble</td>
</tr>
<tr>
<td>#70438h</td>
<td>Size Information for the plug-in</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#7043Dh</td>
<td>Start card in port 2</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70442h</td>
<td></td>
<td>11 nibbles</td>
</tr>
<tr>
<td>#7044Dh</td>
<td>End of Built-in RAM backup zone</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70452h</td>
<td>End of port 1 backup zone</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70457h</td>
<td>End of port 2 backup zone</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#7045Ch</td>
<td>Temporary backup during interrupts</td>
<td>103 nibbles</td>
</tr>
<tr>
<td>#704C3h</td>
<td>Output mask for keyboard test</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>#704C6h</td>
<td></td>
<td>16 nibbles</td>
</tr>
</tbody>
</table>

**Plug-In Cards (HP 48SX)**

This nibble, #700DB, is the same as in the I/O RAM at address #0010Fh:

<table>
<thead>
<tr>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = Port 2 not write-protected</td>
<td>1 = Port 1 not write-protected</td>
<td>1 = Plug-in card present in Port 2</td>
<td>1 = Plug-in card present in Port 1</td>
</tr>
</tbody>
</table>

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:

![Diagram of RS-232C Input Buffer]

The gray area represents the area containing data waiting to be processed.
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)
Key codes stored in the keyboard buffer
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>#704D6h</td>
<td>Machine speed</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#704DBh</td>
<td>Disable keyboard</td>
<td>1 nibble</td>
</tr>
<tr>
<td>#704DCh</td>
<td>Key state</td>
<td>1 nibble</td>
</tr>
<tr>
<td>#704DDh</td>
<td>KeyStart</td>
<td>13 nibbles</td>
</tr>
<tr>
<td>#704EAh</td>
<td>Keyboard</td>
<td>1 nibble</td>
</tr>
<tr>
<td>#704EBh</td>
<td>KeyEnd buffer</td>
<td>1 nibbles</td>
</tr>
<tr>
<td>#704ECH</td>
<td>Key codes</td>
<td>32 nibbles</td>
</tr>
<tr>
<td>#7050Ch</td>
<td>Screen bitmap addr. (#00120h)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70513h</td>
<td>Right margin (#00125h)</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>#70516h</td>
<td>Menu bitmap address (#00130h)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#7051Bh</td>
<td>Menu height (#00128h)</td>
<td>2 nibbles</td>
</tr>
<tr>
<td>#7051Dh</td>
<td>@ of menu GROB</td>
<td>52 nibbles</td>
</tr>
<tr>
<td>#70551h</td>
<td>@ of stack GROB</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70556h</td>
<td>@ of current GROB</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#7055Bh</td>
<td>@ of PICT GROB</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70560h</td>
<td>@ of PICT GROB ?</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#7056Ah</td>
<td>Beginning @ of temporary objects</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#7056Fh</td>
<td>Ending @ of temporary objects</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70574h</td>
<td>Beginning @ of free mem. (B)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>#70579h</td>
<td>Ending @ of free memory (D1)</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Flag (garbage collector)</th>
<th>1 nibble</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>$l_z$ - 6 nibbles</td>
</tr>
<tr>
<td>Object length $l_z$</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>

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  
Flag  
Object  
Length  
Flag  
Object  
Length  
5 nibbles  
1 nibble  
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:

\[
\begin{array}{|c|c|}
\hline
(#7056Fh) & \text{Return address 1} & 5 \text{ nibbles} \\
& \text{Return address 2} & 5 \text{ nibbles} \\
& \vdots & \vdots \\
& \text{Return address } n & 5 \text{ nibbles} \\
\hline
(B) & \\
\end{array}
\]

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:

<table>
<thead>
<tr>
<th>(D1)</th>
<th>Address of object in level 1</th>
<th>5 nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Address of object in level 2</td>
<td>5 nibbles</td>
</tr>
<tr>
<td></td>
<td>Address of the last object</td>
<td>5 nibbles</td>
</tr>
<tr>
<td></td>
<td>00000</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>

To find the address of an object in level \( n \), simply take the value of D1, add \((n-1)\times5\), and read the 5 nibbles at that address. The following assembly program duplicates the SWAP function:

\[
\begin{align*}
A &= DAT1 A \quad * \text{Address of object 1} \\
D1 &= D1 + 5 \quad * \text{Now pointing to level 2} \\
C &= DAT1 A \quad * \text{Address of object 2} \\
DAT1 &= A A \quad * \text{Write address of object 1} \\
D1 &= D1 - 5 \quad * \text{Now pointing to level 1} \\
DAT1 &= C A \quad * \text{Write address of object 2}
\end{align*}
\]

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 ≥ 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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total length L</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Block identifier</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Address of the first local name</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Address of the first contents</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Address of the last local name</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Address of the last contents</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>

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 ("").

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total length L</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Block identifier</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Address of &quot;&quot; (#61D3Ah)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Number of elements on the stack</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Address of &quot;&quot; (#61D3Ah)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Address of the object in level 1</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>Address of &quot;&quot; (#61D3Ah)</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(#7058Dh)</td>
<td>Address of menu label 1</td>
<td>3 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+3h</td>
<td>Address of menu label 2</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+8h</td>
<td>Address of menu label 3</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+Dh</td>
<td>Address of menu label 4</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+12h</td>
<td>Address of menu label 5</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+17h</td>
<td>Address of menu label 6</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+1Ch</td>
<td>Address of menu label 7 (reserved)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+21h</td>
<td>Execution address 1</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+26h</td>
<td>Execution address 2</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+2Bh</td>
<td>Execution address 3</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+30h</td>
<td>Execution address 4</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+35h</td>
<td>Execution address 5</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+3Ah</td>
<td>Execution address 6</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+3Fh</td>
<td>Execution address 7 (reserved)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+44h</td>
<td>Execution address 7 (reserved)</td>
<td>5 nibbles</td>
</tr>
<tr>
<td>(#7058Dh)+49h</td>
<td>Execution address 7 (reserved)</td>
<td>5 nibbles</td>
</tr>
</tbody>
</table>
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).

<table>
<thead>
<tr>
<th>Backup object 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backup object 2</td>
</tr>
<tr>
<td>Last backup object</td>
</tr>
<tr>
<td>00000</td>
</tr>
<tr>
<td>Address</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>#705A6h</td>
</tr>
<tr>
<td>#705ABh</td>
</tr>
<tr>
<td>#705B0h</td>
</tr>
<tr>
<td>#705B5h</td>
</tr>
<tr>
<td>#705BAh</td>
</tr>
<tr>
<td>#705BFh</td>
</tr>
<tr>
<td>#705C4h</td>
</tr>
<tr>
<td>#705C9h</td>
</tr>
<tr>
<td>#705CEh</td>
</tr>
<tr>
<td>#705D3h</td>
</tr>
<tr>
<td>#705D8h</td>
</tr>
<tr>
<td>#705DDh</td>
</tr>
<tr>
<td>#705E2h</td>
</tr>
<tr>
<td>#705E7h</td>
</tr>
<tr>
<td>#705ECH</td>
</tr>
<tr>
<td>#705F1h</td>
</tr>
<tr>
<td>#705F6h</td>
</tr>
<tr>
<td>#705FBh</td>
</tr>
<tr>
<td>#70600h</td>
</tr>
<tr>
<td>#70605h</td>
</tr>
<tr>
<td>#7061Eh</td>
</tr>
<tr>
<td>#70623h</td>
</tr>
<tr>
<td>#70628h</td>
</tr>
<tr>
<td>#70637h</td>
</tr>
<tr>
<td>#7063Ch</td>
</tr>
<tr>
<td>#70641h</td>
</tr>
<tr>
<td>#70646h</td>
</tr>
<tr>
<td>#7064Bh</td>
</tr>
<tr>
<td>#7065Fh</td>
</tr>
<tr>
<td>#70664h</td>
</tr>
<tr>
<td>#70669h</td>
</tr>
<tr>
<td>#7066Eh</td>
</tr>
</tbody>
</table>
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 V AR 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 D9D20888704 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 SOL VR 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 ( שנית), and the third element is if the right shift was pressed first ( ).

<table>
<thead>
<tr>
<th>No.</th>
<th>Menu</th>
<th>Address</th>
<th>No.</th>
<th>Menu</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Last Menu</td>
<td></td>
<td>30</td>
<td>SOLVE.SOLVR</td>
<td>#15200h</td>
</tr>
<tr>
<td>1</td>
<td>CST</td>
<td>#3B239h</td>
<td>31</td>
<td>PLOT</td>
<td>#3BEDB8h</td>
</tr>
<tr>
<td>2</td>
<td>VAR</td>
<td>#3F6D8h</td>
<td>32</td>
<td>PLOT.TYPE</td>
<td>#3C039h</td>
</tr>
<tr>
<td>3</td>
<td>MTH</td>
<td>#3B284h</td>
<td>33</td>
<td>PLOT.PLOTR</td>
<td>#3C0AFh</td>
</tr>
<tr>
<td>4</td>
<td>MTH.PARTS</td>
<td>#3B36Ch</td>
<td>34</td>
<td>ALGEBRA</td>
<td>#3C483h</td>
</tr>
<tr>
<td>5</td>
<td>MTH.PROB</td>
<td>#3B3E4h</td>
<td>35</td>
<td>TIME</td>
<td>#3C4C9h</td>
</tr>
<tr>
<td>6</td>
<td>MTH.HYP</td>
<td>#3B420h</td>
<td>36</td>
<td>TIME.ADJST</td>
<td>#3C671h</td>
</tr>
<tr>
<td>7</td>
<td>MTH.MATR</td>
<td>#3B452h</td>
<td>37</td>
<td>TIME.SET</td>
<td>#3C79Ch</td>
</tr>
<tr>
<td>8</td>
<td>MTH.VECTR</td>
<td>#3B489h</td>
<td>38</td>
<td>TIME.ALM</td>
<td>#3C8D5h</td>
</tr>
<tr>
<td>9</td>
<td>MTH.BASE</td>
<td>#3B4CAh</td>
<td>39</td>
<td>TIME2</td>
<td>#3C9B8h</td>
</tr>
<tr>
<td>10</td>
<td>PRG</td>
<td>#3B542h</td>
<td>40</td>
<td>STAT</td>
<td>#3CAA7h</td>
</tr>
<tr>
<td>11</td>
<td>PRG.STK</td>
<td>#3B622h</td>
<td>41</td>
<td>STAT.MODL</td>
<td>#3CD96h</td>
</tr>
<tr>
<td>12</td>
<td>PRG.OBJ</td>
<td>#3B67Fh</td>
<td>42</td>
<td>UNITS</td>
<td>#3CE5h</td>
</tr>
<tr>
<td>13</td>
<td>PRG.DSPL</td>
<td>#3B6F7h</td>
<td>43</td>
<td>UNITS.LENG</td>
<td>#3D08Ch</td>
</tr>
<tr>
<td>14</td>
<td>PRG.CTRL</td>
<td>#3B7E2h</td>
<td>44</td>
<td>UNITS.AREA</td>
<td>#3D1F3h</td>
</tr>
<tr>
<td>15</td>
<td>PRG.BRCH</td>
<td>#3B8B4h</td>
<td>45</td>
<td>UNITS.VOL</td>
<td>#3D2D6h</td>
</tr>
<tr>
<td>16</td>
<td>PRG.TEST</td>
<td>#3B90Eh</td>
<td>46</td>
<td>UNITS.TIME</td>
<td>#3D451h</td>
</tr>
<tr>
<td>17</td>
<td>PRINT</td>
<td>#3B972h</td>
<td>47</td>
<td>UNITS.SPEED</td>
<td>#3D4B5h</td>
</tr>
<tr>
<td>18</td>
<td>I/O</td>
<td>#3B9A4h</td>
<td>48</td>
<td>UNITS.MASS</td>
<td>#3D53h</td>
</tr>
<tr>
<td>19</td>
<td>I/O.SETUP</td>
<td>#3BA03h</td>
<td>49</td>
<td>UNITS.FORCE</td>
<td>#3D642h</td>
</tr>
<tr>
<td>20</td>
<td>MODES</td>
<td>#3B46h</td>
<td>50</td>
<td>UNITS.ENG</td>
<td>#3D6B5h</td>
</tr>
<tr>
<td>21</td>
<td>MODES2</td>
<td>#3BC8Dh</td>
<td>51</td>
<td>UNITS.POWR</td>
<td>#3D764h</td>
</tr>
<tr>
<td>22</td>
<td>MEMORY</td>
<td>#3BCE7h</td>
<td>52</td>
<td>UNITS.PRES</td>
<td>#3D797h</td>
</tr>
<tr>
<td>23</td>
<td>MEMORY2</td>
<td>#3BD46h</td>
<td>53</td>
<td>UNITS.TEMP</td>
<td>#3D838h</td>
</tr>
<tr>
<td>24</td>
<td>LIBRARY</td>
<td>#3F376h</td>
<td>54</td>
<td>UNITS.ELEC</td>
<td>#3D877h</td>
</tr>
<tr>
<td>25</td>
<td>PORT0</td>
<td>#3BD82h</td>
<td>55</td>
<td>UNITS.ANGL</td>
<td>#3D93Ah</td>
</tr>
<tr>
<td>26</td>
<td>PORT1</td>
<td>#3BDAAh</td>
<td>56</td>
<td>UNITS.LIGHT</td>
<td>#3D9B3h</td>
</tr>
<tr>
<td>27</td>
<td>PORT2</td>
<td>#3BDD2h</td>
<td>57</td>
<td>UNITS.RAD</td>
<td>#3DA42h</td>
</tr>
<tr>
<td>28</td>
<td>EDIT</td>
<td>#3BDFAh</td>
<td>58</td>
<td>UNITS.VISC</td>
<td>#3DABFh</td>
</tr>
<tr>
<td>29</td>
<td>SOLVE</td>
<td>#3BE22h</td>
<td>59</td>
<td>UNITS2</td>
<td>#3DAF2h</td>
</tr>
</tbody>
</table>
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).
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Nibbles</th>
</tr>
</thead>
<tbody>
<tr>
<td>#70673h</td>
<td>Next error to display</td>
<td>5</td>
</tr>
<tr>
<td>#70678h</td>
<td>1 nibble</td>
<td>1</td>
</tr>
<tr>
<td>#70679h</td>
<td>ATTN flag</td>
<td>5</td>
</tr>
<tr>
<td>#7067Eh</td>
<td>33 nibbles</td>
<td></td>
</tr>
<tr>
<td>#7069Fh</td>
<td>Stack size</td>
<td>5</td>
</tr>
<tr>
<td>#706A4h</td>
<td>Random number seed</td>
<td>16</td>
</tr>
<tr>
<td>#706B4h</td>
<td>15 nibbles</td>
<td></td>
</tr>
<tr>
<td>#706C3h</td>
<td>Annunciators</td>
<td>2</td>
</tr>
<tr>
<td>#706C5h</td>
<td>System</td>
<td>16</td>
</tr>
<tr>
<td>#706D5h</td>
<td>User</td>
<td>16</td>
</tr>
<tr>
<td>#706E5h</td>
<td>26 nibbles</td>
<td></td>
</tr>
<tr>
<td>#706FFh</td>
<td>Error number</td>
<td>5</td>
</tr>
<tr>
<td>#70704h</td>
<td>15 nibbles</td>
<td></td>
</tr>
<tr>
<td>#70713h</td>
<td>Prolog</td>
<td>5</td>
</tr>
<tr>
<td>#70718h</td>
<td>Length</td>
<td>5</td>
</tr>
<tr>
<td>#7071Dh</td>
<td>Height (6)</td>
<td>5</td>
</tr>
<tr>
<td>#70722h</td>
<td>Width (10)</td>
<td>5</td>
</tr>
<tr>
<td>#70727h</td>
<td>Pixels</td>
<td>20</td>
</tr>
<tr>
<td>#7073Bh</td>
<td>142 nibbles</td>
<td></td>
</tr>
<tr>
<td>#707C9h</td>
<td>Current menu offset</td>
<td>5</td>
</tr>
<tr>
<td>#707CEh</td>
<td>Last menu offset</td>
<td>5</td>
</tr>
<tr>
<td>#707D3h</td>
<td>6 nibbles</td>
<td></td>
</tr>
<tr>
<td>#707D9h</td>
<td>Number of attached libraries</td>
<td>3</td>
</tr>
<tr>
<td>#707DCh</td>
<td>Number @ of info. First library info</td>
<td>3</td>
</tr>
<tr>
<td>#707DFh</td>
<td>Number @ of info. Last library info</td>
<td>3</td>
</tr>
</tbody>
</table>

**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):

<table>
<thead>
<tr>
<th></th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#706C5h</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>#706C6h</td>
<td>-8</td>
<td>-7</td>
<td>-6</td>
<td>-5</td>
</tr>
<tr>
<td>#706C7h</td>
<td>-12</td>
<td>-11</td>
<td>-10</td>
<td>-9</td>
</tr>
<tr>
<td>#706C8h</td>
<td>-16</td>
<td>-15</td>
<td>-14</td>
<td>-13</td>
</tr>
<tr>
<td>#706C9h</td>
<td>-20</td>
<td>-19</td>
<td>-18</td>
<td>-17</td>
</tr>
<tr>
<td>#706CAh</td>
<td>-24</td>
<td>-23</td>
<td>-22</td>
<td>-21</td>
</tr>
<tr>
<td>#706CBh</td>
<td>-28</td>
<td>-27</td>
<td>-26</td>
<td>-25</td>
</tr>
<tr>
<td>#706CCh</td>
<td>-32</td>
<td>-31</td>
<td>-30</td>
<td>-29</td>
</tr>
<tr>
<td>#706CDh</td>
<td>-36</td>
<td>-35</td>
<td>-34</td>
<td>-33</td>
</tr>
<tr>
<td>#706CEh</td>
<td>-40</td>
<td>-39</td>
<td>-38</td>
<td>-37</td>
</tr>
<tr>
<td>#706CFh</td>
<td>-44</td>
<td>-43</td>
<td>-42</td>
<td>-41</td>
</tr>
<tr>
<td>#706D0h</td>
<td>-48</td>
<td>-47</td>
<td>-46</td>
<td>-45</td>
</tr>
<tr>
<td>#706D1h</td>
<td>-52</td>
<td>-51</td>
<td>-50</td>
<td>-49</td>
</tr>
<tr>
<td>#706D2h</td>
<td>-56</td>
<td>-55</td>
<td>-54</td>
<td>-53</td>
</tr>
<tr>
<td>#706D3h</td>
<td>-60</td>
<td>-59</td>
<td>-58</td>
<td>-57</td>
</tr>
<tr>
<td>#706D4h</td>
<td>-64</td>
<td>-63</td>
<td>-62</td>
<td>-61</td>
</tr>
</tbody>
</table>

### User Flags (1 to 64):

<table>
<thead>
<tr>
<th></th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#706D5h</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>#706D6h</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>#706D7h</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>#706D8h</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>#706D9h</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>#706DAh</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>#706DBh</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>#706DCh</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td>29</td>
</tr>
<tr>
<td>#706DDh</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
</tr>
<tr>
<td>#706DEh</td>
<td>40</td>
<td>39</td>
<td>38</td>
<td>37</td>
</tr>
<tr>
<td>#706DFh</td>
<td>44</td>
<td>43</td>
<td>42</td>
<td>41</td>
</tr>
<tr>
<td>#706E0h</td>
<td>48</td>
<td>47</td>
<td>46</td>
<td>45</td>
</tr>
<tr>
<td>#706E1h</td>
<td>52</td>
<td>51</td>
<td>50</td>
<td>49</td>
</tr>
<tr>
<td>#706E2h</td>
<td>56</td>
<td>55</td>
<td>54</td>
<td>53</td>
</tr>
<tr>
<td>#706E3h</td>
<td>60</td>
<td>59</td>
<td>58</td>
<td>57</td>
</tr>
<tr>
<td>#706E4h</td>
<td>64</td>
<td>63</td>
<td>62</td>
<td>61</td>
</tr>
</tbody>
</table>

14. **RAM**
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

\[ \text{Library number} \times 256 + \text{order number} \]

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.

\section*{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 |, 5 nibbles
- @+Ah Number n, of lines (in pixels) 5 nibbles
- @+Fh Number n, 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 n1 1+1 nibbles
- Last pixels 1+1 nibbles
- @+l, +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:

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Memory Structure</th>
<th>Size (Nibbles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Prolog (02B1E)</td>
<td>5</td>
</tr>
<tr>
<td>@+5h</td>
<td>Total length excluding prolog l,</td>
<td>5</td>
</tr>
<tr>
<td>@+Ah</td>
<td>Number n, of lines (in pixels)</td>
<td>5</td>
</tr>
<tr>
<td>@+Fh</td>
<td>Number n, of columns (in pixels)</td>
<td>5</td>
</tr>
<tr>
<td>@+14h</td>
<td>Object to be created</td>
<td>l, -15</td>
</tr>
<tr>
<td>@+l,+5h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. We 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):

\[
\text{GASS (\#1DB3h)} \leftarrow \text{"GR " OVER SIZE 2 / + " " + SWAP + OBJ} \rightleftharpoons \text{#4017h SYSEVAL #56B6h SYSEVAL DROP NEWOB}
\]

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:

\[
\text{C2R28 B18BB 7556C 6C6B2 46F6E 63682 12}
\]

To code this object, just enter the code as a character string (with no spaces, no new lines): "C2R20B10007556C6C60246F6E6560212"

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: $A = \text{DAT0 } A, D0 = D0 + 5, PC = (A)$ (coded as 142164808C).

- **D1** is the stack pointer. If we execute $A = \text{DAT1 } A$, 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 $A = \text{DAT1 } A$ 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 language 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 $\rightarrow$. 

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,

"CCD20F0000142164800C" BY5

returns "CCD20 F0000 14216 4800C "

BY5 (# 74BAh)
```plaintext
" • " 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  +
0220  CON(5)  PROL_CODE  Code object
08000  CON(5)  (end)-(start)  Code length
8FB9760  GOSBML  SAVE_REG  Bckup regs.

start

143  A=DAT1  A  A-size
130  D0=A  D0=D0+  10
131  D1=A  D1=D1+  5
143  A=DAT1  A
174  D1=D1+  5
818F84  A=A-5  A
819F0  ASRB  A

11
89A9  ?B=0  A  B= # of characters in the string
83  GOYES  14  Done?
14B  A=DAT1  B  Yes --> end!
3103  LCHEX  30  ASCII code for 0
9E2  ?R<C  B  ASCII code for 9
32  GOYES  13  Bad character
3193  LCHEX  39  ASCII code for A
9EA  ?R<C  B  Good character
41  GOYES  12  ASCII code for F
3114  LCHEX  41
9E2  ?R<C  B
11  GOYES  13
3164  LCHEX  46
```

218  PART THREE: LIBRARY OF PROGRAMS
Bad character
Good char --> rewrite
Next
One less
Loop again
Mark the end
with char 00
Restore regs.
Return to RPL

CLEAN (# CD56h)

D9D20 B4E02 76BA1 CCD20 08000 8FB97 60143 13013
11691 74143 17481 8F848 19F0D 89A98 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
3FFB1 CON(5) DROP2 of arguments
CCD28 CON(S) 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

142 A=DAT0 A
340FF7 LCHEX #7FFF0 Read # of nibbles to read
8B6 ?C<A A Maximum size
40 GOYES 10
D6 C=A A Size correct
C6 C=C+C A Size too big—set to max.

8FD7B50 GOSBVL #05B7D Number of nibbles to reserve (2 per character)
132 ADD0ex Reserve
147 C=DAT1 A
134 D0=C
169 D0=D0+ 10
146 C=DAT0 A
D0=address of object in stack level 1

D5 B=C A
174 D1=D1+ 5
147 C=DAT1 A
Read the contents (size to peek)
```

220

PART THREE: LIBRARY OF PROGRAMS
DO=address of object in stack level 2

Read the contents

Done?
Yes --> end

Read one nibble

Transform to ASCII code
(0->'0'=48...
15->'F'=70)

Write into the string

Next character

Next nibble

One less

Loop

Restore regs.

Result -> stack

Return to RPL

Program end

PEEK (# ED02h)

D9D20 2ABF1 3FBF1 CCD20 3A000 8FB97 60147 13416
91423 40FFF 78864 0D6C6 8F07B 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
3FBF1   CON(5)   DROP2
CCD20   CON(5)   PROL_CODE       Code object
48000   CON(5)   (end)-(start)   Code length
8FB9760 GOSBVL   SAVE_REG       Backup regs.
start

143     A=DAT1   A
132     ADD@ex
164     D0=D0+   5
146     C=DAT0   A
164     D0=D0+   5
05      B=C      A
174     D1=D1+   5
143     A=DAT1   A
131     D1=A     
179     D1=D1+   10
143     A=DAT1   A
131     D1=A     

3450000 LCHEX   #00005

/1      
E1      B=B-C   A
8A9     ??=0     A
13      GOYES   13
14A     A=DAT0   B

Done?   Yes -> end
Yes -> end
Read a char
```
Convert ASCII to Hexadecimal

(48='0' -> 0

70='F' -> 15)

Write to memory

Next char

Next nibble

Loop...

Restore regs.

DROP2

Return to RPL

Program end

POKE (# 14A5h)

D9D20 2ABF1 3FBF1 CCD20 48000 8FB97 60143 13216
41461 64051 74143 13117 91431 31345 0000E 18A91
314A3 103B6 A3190 9EA90 3170B 6A159 01611 70342
00006 DCF8F 2D760 179E7 E7142 16480 8CB21 30
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 #78086h #18h HRPEEK (peek at 16 nibbles starting at #70000h in the hidden ROM) will return the character string "D21098FFF8108E78".

**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:

```
09028 CONC(S) PROL_PRGM Program object
2ABF1 CONC(S) DUP2 Verify the number of arguments
3FBF1 CONC(S) DROP2

start

C0108 CONC(S) PROL_CODE Code object
D4100 CONC(S) (end)-(start) Code length
8FB9760 GOSBVL SAVE_REG Backup regs.

147 C=DAT1 A
134 D0=C
169 D0=D0+ 10

142 A=DAT0 A
340FFF7 LCHEX #7FFF0
```

**Program object**

**Verify the number of arguments**

**Code object**

**Code length**

**Backup regs.**

**D0=address of object in stack level 1**

**D0=address of object contents in stack level 1 (PEEK length)**

**Read number of nibbles to be read**

**Maximum size**

---

224 **Part Three: Library of Programs**
Size correct
Size is too big—change to maximum.
No. of nibbles to reserve (2 per character)
Reserve

D0=address of object in stack level 1
Read the contents (size of peek)

D0=address of object in stack level 2
Read the contents

No keyb. int.

Done?

Yes --> end
One less

A=mem. address of 'here'
where is HRPEEK ?
In a plug-in card

C=memory address of '14'
C=address of '14' after displacement of built-in RAM to #F0000h

Displace built-in RAM and call routine found at address in field A of C
6160 GOTO 16
112 A=R2
131 D1=A
AE0 A=0 B
15B0 A=DAT1 I
101 R1=A
01 RTN

340007 LCHEX #70000
804 UNCNFG
340000F LCHEX #F0000
805 CONFIG
3400006 LCHEX #60000
805 CONFIG
112 A=R2
131 D1=A
AE0 A=0 B
15B0 A=DAT1 I
101 R1=A
3400006 LCHEX #60000
804 UNCNFG
340000F LCHEX #F0000
805 CONFIG
3400007 LCHEX #70000
805 CONFIG

8080 INTON
111 A=R1
3103 LCHEX 30
A6A A=A+C B
3193 LCHEX 39
9EA ?C>=A B
90 GOYES 17
3170 LCHEX 07
A6A A=A+C B

11B C=R3
134 D0=C
148 DAT0=A B
161 D0=D0+ 2
136 CD0ex
10B R3=C
11A C=R2
E6 C=C+1 A
10A R2=C
682F GOTO 12
85F ST=1 15
8F2D760 GOSBVL LOAD_REG

Read one nibble from R2 and save it in register R1
Displace the RAM to #60000h
Read one nibble
Return RAM to #70000h
Interrupts OK
Convert the nibble read to ASCII
Write
Next!
Loop
Restore regs.

PART THREE: LIBRARY OF PROGRAMS
174  D1=D1+  5  ;
E7   D=D+1   A  ;DROP
118  C=R0
145  DAT1=C   A
142  A=DAT0  A
164  D8=D8+  5
808C  PC=(A)
  end  B2130  CON(5)  EPILOG  Program end

Resulting string on stack
Return to RPL

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  C80BF  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  39E99  03170  A6A11  B1341  48161  13610  B11AE
610A6  82F85  F8F2D  76017  4E711  81451  42164  808CB
2130
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 the address will be
A0000  CON(5)  #0000A
00000  CON(5)  #00000
CB2A1  CON(5)  NEWOB
DBBF1  CON(5)  SWAP
CCD20  CON(5)  PROL_CODE  Code object

start 62000  CON(5)  (end)-(start)  Code length
147  C=DAT1  A
174  D1=D1+  5  Remove object from stack
E7  D=D+1  A
143  A=DAT1  A
133  AD1ex
179  D1=D1+  10
145  DAT1=C  A  Write @
131  D1=A
142  A=DAT0  A
164  D0=D0+  5
808C  PC=(A)  Return to RPL
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
```

PART THREE: LIBRARY OF PROGRAMS
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 "C2A20B000132333" 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
>

229
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

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  D1=string address
137   CD1ex  A  A=string length
109   R1=C  Empty string?
174   D1=D1+  5  Yes --> end
143   A=DAT1  A  Number of codes
3450000  LCHEX  #00005
8A2   ?C=A  A
57    GOYES  15
EA    A=A-C  A
81C   ASRB
103   R3=A
174   D1=D1+  5
137   CD1ex
10A   R2=C
D6    C=A  A
84A   ST=0  10

l1 8F8DA68  GOSBVL  #06AD8  Reserve memory
501   GONC  12  Ok!
8FD3361  GOSBVL  #1633D  Garbage collector
11B   C=R3
6BEEF  GOTO  11

l2 119   C=R1
135   D1=C
132   AD0ex
141   DAT1=A  A  Object reserved on
130   D0=A  the stack
113   A=R3
D8    B=A  A
CD    B=B-1  A
11A   C=R2
135   D1=C
```
I3 14B  A=DAT1  B
3103  LCHEX  #30
B6A  A=A-C  B
3190  LCHEX  #09
9EA  ?C>=A  B
90  G0YES  14
3170  LCHEX  #07
B6A  A=A-C  B

I4 1580  DAT0=A  1
160  D0=D0+  1
171  D1=D1+  2
CD  B=B-1  A
59D  GONC  13

I5 8F2D760  GOSBVL  LOAD_REG
142  A=DAT0  A
164  D0=D0+  5
808C  PC=(A)
end B2130  CON(5)  EPILOG

read one code

ASCII Code

-> hexadecimal

Write

One less
Continue if necessary

Restore regs.
Return to RPL

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 #1A8262h CHK

<table>
<thead>
<tr>
<th>Prolog</th>
<th>Object Type</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>02911</td>
<td>Any Object</td>
<td>00</td>
</tr>
<tr>
<td>02933</td>
<td>System Binary</td>
<td>01</td>
</tr>
<tr>
<td>02955</td>
<td>Real Number</td>
<td>02</td>
</tr>
<tr>
<td>02977</td>
<td>Long Real</td>
<td>03</td>
</tr>
<tr>
<td>0299D</td>
<td>Complex Number</td>
<td>04</td>
</tr>
<tr>
<td>029BF</td>
<td>Long Complex</td>
<td>05</td>
</tr>
<tr>
<td>029E8</td>
<td>Character</td>
<td>06</td>
</tr>
<tr>
<td>02A0A</td>
<td>Array</td>
<td>07</td>
</tr>
<tr>
<td>02A2C</td>
<td>Linked Array</td>
<td>08</td>
</tr>
<tr>
<td>02A4E</td>
<td>String</td>
<td>09</td>
</tr>
<tr>
<td>02A74</td>
<td>Binary Integer</td>
<td>0A</td>
</tr>
<tr>
<td>02A96</td>
<td>List</td>
<td>0B</td>
</tr>
<tr>
<td>02AB8</td>
<td>Directory</td>
<td>0C</td>
</tr>
<tr>
<td>02ADA</td>
<td>Algebraic Object</td>
<td>0D</td>
</tr>
<tr>
<td>02AFC</td>
<td>Unit Object</td>
<td>0E</td>
</tr>
<tr>
<td>02B1E</td>
<td>Tagged Object</td>
<td>0F</td>
</tr>
<tr>
<td></td>
<td>Graphic Object</td>
<td>10</td>
</tr>
</tbody>
</table>


![Image: CHK](image)

---

*Part Three: Library of Programs*
Here is the commented assembly source listing for CHK:

```
CCD20   CON(5)  PROL_CODE     Code object
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  Maximum arguments
3400000 LCHEX #00008 C(W)=types
D5      B=C     A
147     C=DAT1 A
134     D0=C
169     D0=D0+ 10
1567    C=DAT0 W
174     D1=D1+ 5
143     A=DAT1 A
130     D0=A
169     D0=D0+ 10
1527    A=DAT0 W
174     D1=D1+ 5
888     ?A>B     A  More than 8 args?
71      GOYES err1 Yes --> error
```
PART THREE: LIBRARY OF PROGRAMS
Backup

Type OK?

No --> error

Get a prolog

End of stack --> error

(Too Few Arguments)

A = object prologue

Any object?

Yes --> OK

Prologue OK?

Yes --> 15

Obj. prologue required

--> "Bad Argument Type"

One less

Next type

Next object

Loop

end

CHK (# FD?Ch)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>134</td>
<td>D0=C</td>
</tr>
<tr>
<td>06</td>
<td>RSTK=C</td>
</tr>
<tr>
<td>3101</td>
<td>LCHEX</td>
</tr>
<tr>
<td>9E7</td>
<td>?D&lt;C</td>
</tr>
<tr>
<td>60</td>
<td>GOYES</td>
</tr>
<tr>
<td>693F</td>
<td>GOTO</td>
</tr>
<tr>
<td>96B</td>
<td>?D=0</td>
</tr>
<tr>
<td>60</td>
<td>GOYES</td>
</tr>
<tr>
<td>147</td>
<td>C=DAT1</td>
</tr>
<tr>
<td>8AE</td>
<td>?C#=0</td>
</tr>
<tr>
<td>00</td>
<td>GOYES</td>
</tr>
<tr>
<td>3410200</td>
<td>LCHEX</td>
</tr>
<tr>
<td>6E1F</td>
<td>GOTO</td>
</tr>
<tr>
<td>137</td>
<td>CD1EX</td>
</tr>
<tr>
<td>143</td>
<td>A=DAT1</td>
</tr>
<tr>
<td>135</td>
<td>D1=C</td>
</tr>
<tr>
<td>146</td>
<td>C=DAT0</td>
</tr>
<tr>
<td>21</td>
<td>GOYES</td>
</tr>
<tr>
<td>8A2</td>
<td>?A=C</td>
</tr>
<tr>
<td>00</td>
<td>GOYES</td>
</tr>
<tr>
<td>3420200</td>
<td>LCHEX</td>
</tr>
<tr>
<td>6DFE</td>
<td>GOTO</td>
</tr>
<tr>
<td>CD</td>
<td>B=B-1</td>
</tr>
<tr>
<td>BF7</td>
<td>DSR</td>
</tr>
<tr>
<td>174</td>
<td>D1=D1+</td>
</tr>
<tr>
<td>689F</td>
<td>GOTO</td>
</tr>
</tbody>
</table>

235
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
86000  CON(5)   (end)-(start) Code length
8FB9760 COSBYL   SAVE_REG   Backup regs.

143    A=DAT1  A
131    D1=A
174    D1=D1+  5
137    CD1ex
135    D1=C
143    A=DAT1  A

C2     C=C+A  A
134    D0=C
174    D1=D1+  5

181    D0=D0-  2

818F84  A=A-5  A
8A8     ?A=0   A
52     GOYES  12

14B    A=DAT1  B
14E    C=DAT0  B
14D    DAT1=C  B
148    DAT0=A  B
171    D1=D1+  2
181    D0=D0-  2
133    AD1ex
131    D1=A
136    CD0ex
134    D0=C
```

**PART THREE: LIBRARY OF PROGRAMS**
8BA ?C>=A A Again?
FD GOYES 11
12 8F2D760 GOSBYL 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 FO550 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 difficult 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.
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 \(\text{ON}+\) and \(\text{ON}-\) methods, which offer values from #3h to #13h.

CONTRAST(# 7BF1h)

\[
\text{HEX} \ # \ 101h \ \text{OVER} \ # \ Fh \ \text{AND} \ \rightarrow \text{STR} \ 3 \ 3 \ \text{SUB} \ "\#" \\
\# \ 102h \ # \ 1h \ \text{PEEK} \ + \ \text{STR} \rightarrow \ # \ Eh \ \text{AND} \ 4 \ \text{ROLL} \ 16 \\
/ \ # \ 1h \ \text{AND} \ \text{OR} \ \rightarrow \text{STR} \ 3 \ 3 \ \text{SUB} \ + \ \text{POKE}
\]

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 (\(\text{ON} \rightarrow \text{C}\)).

DISPON(# 10B7h)

\[
\# \ 100h \ "\#" \ \text{OVER} \ # \ 1h \ \text{PEEK} \ + \ \text{STR} \rightarrow \ # \ 8h \ \text{OR} \\
\rightarrow \text{STR} \ 3 \ 3 \ \text{SUB} \ \text{POKE}
\]

DISPOFF(# 8EF6h)

\[
\# \ 100h \ "\#" \ \text{OVER} \ # \ 1h \ \text{PEEK} \ + \ \text{STR} \rightarrow \ # \ 7h \ \text{AND} \\
\rightarrow \text{STR} \ 3 \ 3 \ \text{SUB} \ \text{POKE}
\]
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
```

PART THREE: LIBRARY OF PROGRAMS
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
  "   STO

DO 'I' STO L READ 1 + GET EVAL + STOS
UNTIL P $ >
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" "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

244

Part Three: Library of Programs
CODE a "=" a z b + + + + SPC2 +
IF
  15 ==
THEN
 "A"
ELSE
  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 C6 }

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 +
>

245
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 MOD 1 + GET
SPC2
IF
  \ x \ 4 ==
THEN
  IF
    \ y \ 8 <
    THEN
      "A"
    ELSE
      "B"
    END
  ELSE
    INC READ
    \ z
    <
      IF
        \ y \ 8 <
        THEN
```

Part Three: Library of Programs
ELSE
READ 1 \rightarrow STR
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)
\[ \text{INC CODE "P=" SPC2 READ \rightarrow STR + } \]

A31(#6DCAh)
\[ \text{INC READ + \times + SPC2 Z INC P DUP } \times \text{ + DUP 'P' STO SUB}
  \text{REVERSE + } \]
\[ \text{ } \]
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 (# 4B81h)

INC TAKE INC TAKE + DUP IF "00" == THEN DROP "RTNNC" ELSE "GONC" SWAP # 100h 1 SAUTREL END CODE SWAP

Part Three: Library of Programs
A6(# A19Ch)

\[
\begin{align*}
&\text{Z INC P DUP 3 + SUB DUP IF} \\
&\quad 1 \ 3 \ \text{SUB "300" ==} \\
&\quad \text{THEN} \\
&\quad \text{DROP "NOP4"} \\
&\quad \text{ELSE} \\
&\quad \text{DUP IF} \\
&\quad \quad "4000" == \\
&\quad \quad \text{THEN} \\
&\quad \quad \text{DROP "NOP5" INC} \\
&\quad \quad \text{ELSE} \\
&\quad \quad 1 \ 3 \ \text{SUB "GOTO" SWAP # 1000h 1 SAUTREL} \\
&\quad \quad \text{END} \\
&\quad \text{END} \\
&\text{INC INC CODE SWAP}
\end{align*}
\]

M(# CC5Ch)

\[
\begin{align*}
\{ \ B0 \ B1 \ B1 \ B3 \ B4 \ B4 \ B6 \ B6 \\
B6 \ B6 \ BA \ BA \ BC \ BC \ BC \ BC \ BC \}
\end{align*}
\]

B1(# 9732h)

\[
\begin{align*}
&\text{"U" \ TAKE + STR
\to\ INC \ READ \ 1 \ + \ \text{GET \ EVAL \ CODE \ SWAP}}
\end{align*}
\]

B3(# FA87h)

\[
\begin{align*}
&\text{B1 \ GOYES}
\end{align*}
\]

B4(# 5589h)

\[
\begin{align*}
&\{ \ "ST=0" \ "ST=1" \} \ \text{READ 3 \ - \ \text{GET \ INC \ SPC2 \ READ} \\
&\to\ \text{STR + CODE \ SWAP}
\end{align*}
\]
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" }
\( BA(\#\ 2958h) \)
\[ \text{READ INC READ} \]
\[ \rightarrow x \ y \]
\[ \text{CODE} \]
\[ \text{IF} \]
\[ x \ 10 == \]
\[ \text{THEN} \]
\[ A \]
\[ \text{ELSE} \]
\[ B \]
\[ \text{END} \]
\[ y \ 1 + \text{GET SPC2} + "A" \text{ GOYES} \]

\( BC(\#\ 2CCCh) \)
\[ \{ "GOLONG" 4 "GOYLNG" 5 "GOSUBL" 4 "GOSBYL" 5 \} \]
\[ \text{READ} \ 2 * 23 - \text{DUP} \ 1 + \text{SUB} \text{ LIST} \rightarrow \text{DROP} \]
\[ \rightarrow \ a \ \ b \]
\[ \text{IF} \]
\[ a \ Z \ P \ 1 + \text{DUP} \ b \ 1 - \text{SUB} \]
\[ \text{THEN} \]
\[ b \ 5 == \]
\[ \text{SWAP} \ \text{SPC2} \ \text{SWAP} \ \text{REVERSE} + \]
\[ \text{ELSE} \]
\[ \#\ \text{10000h} \ 2 \ \text{READ} \ 14 == 4 * + \ \text{SAUTREL} \]
\[ \text{END} \]
\[ \text{P} \ \text{b} + 'P' \ \text{STO} \ \text{CODE} \ \text{SWAP} \]

\( V0(\#\ E524h) \)
\[ \{ "INTON" \ V01 \ V02 \ "BUSCB" \ V04 \ V04 \ V04 \ V04 \ V04 \ V04 \ "PC=(A)" \ "BUSCD" \ "PC=(C)" \ "INTOFF" \} \]
U18 (# 8795h)

```
READ 8 == INC READ INC READ 1 +

RA r GET
IF

THEN
DUP "=" SWAP ++
IF

r 8 <

THEN
"+
ELSE
"-

END
+ INC READ 1 + +
ELSE
"SRB" +
END
f CHA
```

V08 (# 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" "CSDL" "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 "XH=0" "SB=0" 3 "SR=0" 5 6 7 "MP=0" 9 10
  11 12 13 14 "CLRHOST" }

U3 (# EA2Ch)
{ 0 "?XM=0" "?SB=0" 3 "?SR=0" 5 6 7 "?MP=0" }

A9 (# 40ADh)
< A B NORMAL GOYES >

AA (# 2CB0h)
< C D NORMAL >

AB (# B167h)
< 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=0" "?B=0" "?C=0" "?D=0" "?A=0" "?B=0"
  "?C=0" "?D=0" }
\[
C \ (\# \ 50A9h) \ \\
\{ \ "A=A+B" \ "B=B+C" \ "C=C+A" \ "D=D+C" \ "A=A+A" \ "B=B+B" \\
\ "C=C+C" \ "D=D+D" \ "B=B+A" \ "C=C+B" \ "A=A+C" \ "C=C+D" \\
\ "A=A-1" \ "B=B-1" \ "C=C-1" \ "D=D-1" \} \\
\]

\[
D \ (\# \ 9930h) \ \\
\{ \ "A=0" \ "B=0" \ "C=0" \ "D=0" \ "A=B" \ "B=C" \ "C=A" \ "D=C" \\
\ "B=A" \ "C=B" \ "A=C" \ "C=D" \ "ABex" \ "CBex" \ "CAex" \\
\ "CDex" \} \\
\]

\[
E \ (\# \ C345h) \ \\
\{ \ "A=A-B" \ "B=B-C" \ "C=C-A" \ "D=D-C" \ "A=A+1" \ "B=B+1" \\
\ "C=C+1" \ "D=D+1" \ "B=B-A" \ "C=C-B" \ "A=A-C" \ "C=C-D" \\
\ "A=B-A" \ "B=C-B" \ "C=A-C" \ "D=C-D" \} \\
\]

\[
F \ (\# \ 7B66h) \ \\
\{ \ "ASL" \ "BSL" \ "CSL" \ "DSL" \ "ASR" \ "BSR" \ "CSR" \ "DSR" \\
\ "A=-A" \ "B=-B" \ "C=-C" \ "D=-D" \ "A=-A-1" \ "B=-B-1" \\
\ "C=-C-1" \ "D=-D-1" \} \\
\]

\[
SPC \ (\# \ EA19h) \quad (7 \ spaces) \\
\]

\[
SPC1 \ (\# \ DF86h) \\
\{ \ SPC \ 1 \ 7 \ 4 \ PICK \ SIZE \ - \ SUB \ + \ " \ " \ + \} \\
\]

\[
SPC2 \ (\# \ DF86h) \\
\{ \ SPC \ 1 \ 7 \ 4 \ PICK \ SIZE \ - \ SUB \ + \ " \ " \ + \} \\
\]

\[
ADRSTR \ (\# \ 1EF0h) \\
\{ \ # \ 100000h \ + \ STR \ 4 \ 8 \ SUB \} \\
\]

\[
CODE \ (\# \ A7D6h) \\
\{ \ ADR \ I \ 1 \ - \ + \ ADRSTR \ " \ " \ + \ Z \ I \ P \ SUB \ SPC1 \ + \} \\
\]
Sautrel (# D63Eh)

\[
\begin{align*}
\text{SPC2 ADR I} + 1 - c &+ "#" \text{ a REVERSE + OBJ} \to \text{DUP IF} \\
&b 2 < \text{ THEN} \\
&\text{ELSE} \\
&b \text{ SWAP --} \\
&\text{END} \\
&\text{ADRSTR +}
\end{align*}
\]

Goyes (# E103h)

\[
\begin{align*}
\text{INC P 'I' STO TAKE INC TAKE +} \\
&→ a \\
&\text{10 CHR CODE IF} \\
&a "00" \text{ THEN} \\
&"RTNYES" \\
&\text{ELSE} \\
&"GOYES" a \# 100h 0 \text{ SAUTREL} \\
&+ + \\
&\text{END}
\end{align*}
\]

Normal (# B551h)

\[
\begin{align*}
\text{INC READ INC READ} \\
&→ x y \\
&\text{CODE IF} \\
&y 8 < \text{ THEN}
\end{align*}
\]
```
ELSE
END
y 1 + GET SPC2 x CH +
```
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 (# B498h)
< 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 + # 1FFFFFFh AND A→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 /
    >
  >
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)
\[
\text{\(\rightarrow\) MOTIF AD FIN PRGM}
\]
\[
\text{\# 100h DUP MOTIF SIZE \(\rightarrow\) LEN LENP}
\]
\[
\{ \}
\]
DO
AD DUP 1 DISP LENP PRGM EVAL
IF
MOTIF POS AD OVER THEN
+ DUP 'AD' STO 1 - DUP IF
FIN \(\geq\)
THEN
DROP
ELSE
DUP 2 DISP 1000 .07 BEEP + END
ELSE
+ LEN + 'AD' STO END
UNTIL
AD FIN \(\geq\)
END

ROMSEARCH(# 5E4Eh)
\[
\text{\(\rightarrow\) MOTIF}
\]
\[
\text{MOTIF \# 0h \# 70000h \text{'PEEK'} SEARCH MOTIF}
\]
\[
\text{\# 70000h \# 80000h \text{'HRPEEK'} SEARCH +}
\]

RAMSEARCH(# 88ABh)
\[
\text{\# 70000h \# 70669h \# 5h PEEK STR\(\rightarrow\)A \text{'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 GRASS) 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  8F207
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.
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
beginB6000 CON(5) (end)-(begin)
8FB9760 GOSBYL SAVE_REG
143 A=DAT1 A
132 ADDex

164 D0=D0+ 5
3450000 LCHEX #00005
142 A=DAT0 A
EA A=A-C A
D8 B=A A
164 D0=D0+ 5
174 D1=D1+ 5
143 A=DAT1 A
133 ADDex

174 D1=D1+ 5
147 C=DAT1 A
133 ADDex
C2 C=C+A A
137 ADDex
3103 LCHEX #30

I1 8A9 ?B=0 A
61 GOYES 12
1C1 D1=D1- 2
15E0 C=DAT0 1
15D1 DAT1=C 2
160 D0=D0+ 1
CD B=B-1 A
6AEF GOTO 11

I2 8F2D760 GOSBYL LOAD_REG
142 A=DAT0 A
164 D0=D0+ 5
808C PC=(A)
```

Code object
Code Length
Backup regs.

D0=address of object in stack level 1
Object length

D1=address of object in stack level 2

Done?
Yes --> end

Read a digit

One digit less

Restore regs.
Return to RPL
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.

```
begin
76000  CON(5)  PROL_CODE  Code object
8FB9760  GOSBVL  SAVE_REG  Code length
143    A=DAT1  A  Backup regs.
132    AD0ex
164    D0=D0+  5  D0=Address of object
3450000  LCHEX  #00005  in stack level 1
142    A=DAT0  A
EA     A=A-C  A
D8     B=A  A
164    D0=D0+  5  D1=Address of object
174    D1=D1+  5  in stack level 2
143    A=DAT1  A
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    GOSYES  12  Yes --> end
1C1    D1=D1-  2
15B0    A=DAT1  1
1500    DAT0=A  P
160    D0=D0+  1
CD     B=B-1  A  One digit less
6AEF   GOTO 1  1  Loop

l2
8F2D760  GOSBVL  LOAD_REG  Restore regs.
142    A=DAT0  A  Return to RPL
164    D0=D0+  5
808C    PC=(A)
end
```

PART THREE: LIBRARY OF PROGRAMS
This program will remove any leading zeros from an integer (convert "00123" to "123", for example).

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>CCD20</td>
<td>CON(5) PROL_CODE</td>
</tr>
<tr>
<td>02</td>
<td>begin5E000</td>
<td>CON(5) (end)-(begin)</td>
</tr>
<tr>
<td>03</td>
<td>8FB9760</td>
<td>GOSBVL SAVE_REG</td>
</tr>
<tr>
<td>04</td>
<td>143</td>
<td>A=DAT1 A</td>
</tr>
<tr>
<td>05</td>
<td>130</td>
<td>D0=A</td>
</tr>
<tr>
<td>06</td>
<td>169</td>
<td>D0=D0+ 10</td>
</tr>
<tr>
<td>07</td>
<td>174</td>
<td>D1=D1+ 5</td>
</tr>
<tr>
<td>08</td>
<td>143</td>
<td>A=DAT1 A</td>
</tr>
<tr>
<td>09</td>
<td>131</td>
<td>D1=A</td>
</tr>
<tr>
<td>10</td>
<td>174</td>
<td>D1=D1+ 5</td>
</tr>
<tr>
<td>11</td>
<td>143</td>
<td>A=DAT1 A</td>
</tr>
<tr>
<td>12</td>
<td>818F84</td>
<td>A=A-5 A</td>
</tr>
<tr>
<td>13</td>
<td>172</td>
<td>D1=D1+ 3</td>
</tr>
<tr>
<td>14</td>
<td>D3</td>
<td>D=0 A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>l1</td>
<td>D1=D1+ 2</td>
</tr>
<tr>
<td></td>
<td>E7</td>
<td>D=D+1 A</td>
</tr>
<tr>
<td></td>
<td>818F81</td>
<td>A=A-2 A</td>
</tr>
<tr>
<td></td>
<td>8AF8</td>
<td>?A=0 A</td>
</tr>
<tr>
<td></td>
<td>D8</td>
<td>GOYES 12</td>
</tr>
<tr>
<td></td>
<td>1570</td>
<td>C=DAT1 P</td>
</tr>
<tr>
<td></td>
<td>90A</td>
<td>?C=0 P</td>
</tr>
<tr>
<td></td>
<td>9E</td>
<td>GOYES 11</td>
</tr>
<tr>
<td></td>
<td>l2</td>
<td>C=D A</td>
</tr>
<tr>
<td></td>
<td>DB</td>
<td>DAT0=C A</td>
</tr>
<tr>
<td></td>
<td>8F2D760</td>
<td>GOSBVL LOAD_REG</td>
</tr>
<tr>
<td></td>
<td>142</td>
<td>A=DAT0 A</td>
</tr>
<tr>
<td></td>
<td>164</td>
<td>D0=D0+ 5</td>
</tr>
<tr>
<td></td>
<td>888C</td>
<td>PC=(A)</td>
</tr>
</tbody>
</table>

Code object
- Code length
- Backup regs.

D0=Address of object in stack level 1
D1=Address of object in stack level 2
Object length
Number of zeroes to remove

Done?
Yes --> end
A zero?
Yes --> loop
write the number of zeros to remove
Restore regs.
Return to RPL
This program sets the integer given as an argument to zero, in the special integer format.

```
CCD20 CON(5) PROL_CODE
begin 54000 CON(5) (end)-(begin) Code object
  8FB9760 GOSBVL SAVE_REG Code length
  143 A=DAT1 A Backup regs.
  131 D1=A

  174 D1=D1+ 5 D1=Address of object
  143 A=DAT1 A in stack level 1
  174 D1=D1+ 5
  C4 A=A+A A
  F4 ASR A

  AF2 C=0 W A=number of 8-digit
  l1 8A8 ?A=0 A blocks for this object
   F0 G0YES 12 Done?
   15D7 DAT1=C 8 Yes --> end
   177 D1=D1+ 8 Set to zero
   CC A=A-1 A Loop
   61FF GOTO 11

  l2 8F2D760 GOSBVL LOAD_REG
  142 A=DAT0 A Restore regs.
  164 D0=D0+ 5 Return to RPL.
  808C PC=(A)

end
```
ADD.LM

This program will add two integers. It works with blocks of 8 digits.

<table>
<thead>
<tr>
<th>CCD20</th>
<th>CON(5)</th>
<th>PROL_CODE</th>
<th>Code object</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>57000</td>
<td>CON(5)</td>
<td>(end)-(begin)</td>
</tr>
<tr>
<td>8FB9760</td>
<td>GOSBYVL</td>
<td>SAVE_REG</td>
<td>Code length</td>
</tr>
<tr>
<td>143</td>
<td>A=DAT1</td>
<td>A</td>
<td>Backup regs.</td>
</tr>
<tr>
<td>130</td>
<td>D8=A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>169</td>
<td>D8=D8+</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>D1=D1+</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>A=DAT1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>D1=A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>D1=D1+</td>
<td>5</td>
<td>D0=Address of object in stack level 1</td>
</tr>
<tr>
<td>147</td>
<td>C=DAT1</td>
<td>A</td>
<td>D1=Address of object in stack level 2</td>
</tr>
<tr>
<td>C6</td>
<td>C=C+C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td>CSR</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D7</td>
<td>D=C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>D1=D1+</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>AF0</td>
<td>A=0</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>P=</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

l1
8AB | ?D=0 | A |
| F2 | GOYES | 12 |
| AF2 | C=0 | W |
| 80F0 | CPex | 0 |
| 05 | SETDEC | |
| 15A7 | A=DAT0 | 8 |
| A72 | C=C+A | W |
| 15B7 | A=DAT1 | 8 |
| A72 | C=C+A | W |
| 04 | SETHEX | |
| 15D7 | DAT1=C | 8 |
| 167 | D8=D8+ | 8 |
| 177 | D1=D1+ | 8 |
| CF | D=D-1 | A |
| 80D8 | P=C | 8 |
| 61DF | GOTO | 11 |

l2
8F2D760 | GOSBYVL | LOAD_REG |
| 142 | A=DAT0 | A |
| 164 | D8=D8+ | 5 |
| 888C | PC=(A) | |

end
This program will subtract two integers. It works with blocks of 8 digits.

SUB.LM

<table>
<thead>
<tr>
<th>Code object</th>
<th>Code length</th>
<th>Backup regs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCD20 CON(5)</td>
<td>PROL_CODE (end)-(begin)</td>
<td>beginning 67000 CON(5)</td>
</tr>
<tr>
<td>8FB9760 GOSBVL</td>
<td>SAVE_REG</td>
<td>143 A=DAT1 A</td>
</tr>
<tr>
<td>130 D0=A</td>
<td>169 D0=D0+ 10</td>
<td>174 D1=D1+ 5</td>
</tr>
<tr>
<td>143 A=DAT1 A</td>
<td>131 D1=A</td>
<td>174 D1=D1+ 5</td>
</tr>
<tr>
<td>147 C=DAT1 A</td>
<td>C6 C=C+C A</td>
<td></td>
</tr>
<tr>
<td>F6 CSR A</td>
<td>D7 D=C</td>
<td></td>
</tr>
<tr>
<td>174 D1=D1+ 5</td>
<td>AF0 A=0 W</td>
<td>l1</td>
</tr>
<tr>
<td>8AB ?D=0 A</td>
<td>23 GOYES 12</td>
<td></td>
</tr>
<tr>
<td>AF2 C=0 W</td>
<td>15E7 C=DAT0 8</td>
<td>05 SETDEC</td>
</tr>
<tr>
<td>A7A A=R+C W</td>
<td>15F7 C=DAT1 8</td>
<td>04 SETHEX</td>
</tr>
<tr>
<td>B72 C=C-A W</td>
<td>15D7 DAT1=C 8</td>
<td>l2</td>
</tr>
<tr>
<td>177 D1=D1+ 8</td>
<td>167 D0=D0+ 8</td>
<td>8F2D760 GOSBVL LOAD_REG</td>
</tr>
<tr>
<td>CF D=D-1 A</td>
<td>142 A=DAT0 A</td>
<td>164 D0=D0+ 5</td>
</tr>
<tr>
<td>AF0 A=0 W</td>
<td>94A ?C=0 S</td>
<td>6ECF GOTO 11</td>
</tr>
<tr>
<td>4D GOYES 11</td>
<td>B64 A=A+1 B</td>
<td>l2</td>
</tr>
<tr>
<td>6EDF GOTO 11</td>
<td>808C PC=(A)</td>
<td>end</td>
</tr>
</tbody>
</table>

D0=Address of object in stack level 1

D1=Address of object in stack level 2

D= # of blocks in obj.

No carry

Done?

Yes --> end

Read 1 block

Decimal mode

Add to carry

Block to subtract

Subtraction

Hexadecimal mode

Write result

One block less

Carry?

No --> loop

Save the carry

Loop

Restore regs.

Return to RPL

PART THREE: LIBRARY OF PROGRAMS
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.

<table>
<thead>
<tr>
<th>Code object</th>
<th>Code length</th>
<th>Backup regs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1=address of contents of level-1 object (the result)</td>
<td>D1=Address of object in stack level 2</td>
<td>Number of blocks of integer in level 2</td>
</tr>
<tr>
<td>R3=address of contents of level-2 obj.</td>
<td>D1=address of object in stack level 3</td>
<td></td>
</tr>
</tbody>
</table>

```
begin
end)
```

```
8FB9760 GOSBVL SAVE_REG
143 A=DAT1 A
818F09 A=A+10 A
101 R1=A
174 D1=D1+ 5
143 A=DAT1 A
133 ADlex
174 D1=D1+ 5
147 C=DAT1 A
818FA4 C=C-5 A
BF2 CSL W
BF2 CSL W
BF2 CSL W
AD7 D=C M
174 D1=D1+ 5
133 ADlex
103 R3=A
174 D1=D1+ 5
143 A=DAT1 A
131 D1=A
174 D1=D1+ 5
147 C=DAT1 A
818FA4 C=C-5 A
BF2 CSL W
BF2 CSL W
BF2 CSL W
174 D1=D1+ 5
133 ADlex
```
R2 = address of object
3 contents

More work?
Yes --> continue
No --> stop

Read a digit

One less digit

Mult by zero?
Yes --> done

Again?
No --> write final carry
And loop
Backup C

Decimal mode

Multiplication

Add the carry
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15A0</td>
<td>A = DAT0</td>
<td>Add to existing</td>
</tr>
<tr>
<td>A62</td>
<td>C = C + A</td>
<td>Write result</td>
</tr>
<tr>
<td>15C0</td>
<td>DAT0 = C</td>
<td>Hexadecimal mode</td>
</tr>
<tr>
<td>04</td>
<td>SETHEX</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>D0 = D0 + 1</td>
<td>Update carry</td>
</tr>
<tr>
<td>170</td>
<td>D1 = D1 + 1</td>
<td>Loop</td>
</tr>
<tr>
<td>A5D</td>
<td>B = B - 1</td>
<td>Restore regs.</td>
</tr>
<tr>
<td>BE6</td>
<td>CSR</td>
<td></td>
</tr>
<tr>
<td>6BAF</td>
<td>GOTO 13</td>
<td>Return to RPL</td>
</tr>
<tr>
<td>8F2D760</td>
<td>Gosbyl LOAD_REG</td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>A = DAT0 A</td>
<td></td>
</tr>
<tr>
<td>164</td>
<td>D0 = D0 + 5</td>
<td></td>
</tr>
<tr>
<td>800C</td>
<td>PC = (A)</td>
<td></td>
</tr>
</tbody>
</table>

**end**
**DIV.LM**

This program divides two integers and returns the integer part of the result.

<table>
<thead>
<tr>
<th>Code object</th>
<th>Code length</th>
<th>Backup regs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCD20 CON(5)</td>
<td>CON(5)</td>
<td>PROL_CODE (end)-(begin)</td>
</tr>
</tbody>
</table>

| 76100 | 8FB9760 | A=DAT1 | 143 | 130 | D0=A |
| 164 | D0=D0+ | 5 |
| 142 | A=DAT0 | A |
| 818F84 | A=A-5 | A |
| 819F0 | ASRB | A |
| 103 | R3=A |
| 164 | D0=D0+ | 5 |
| 132 | AD0ex |
| 102 | R2=A |
| 174 | D1=D1+ | 5 |
| 143 | A=DAT1 | A |
| 818F07 | A=A+8 | A |
| D8 | B=A |
| 174 | D1=D1+ | 5 |
| 143 | A=DAT1 | A |
| 130 | D0=A |
| 167 | D0=D0+ | 8 |
| 174 | D1=D1+ | 5 |
| 143 | A=DAT1 | A |
| 818F04 | A=A+5 | A |
| 131 | D1=A |
| 147 | C=DAT1 | A |
| CA | A=A+C | A |
| 818F81 | A=A-2 | A |
| 100 | R0=A |
| 818FA4 | C=C-5 | A |
| 819F2 | CSRB | A |
| 109 | R1=C |
| AC3 | D=0 | S |
| 113 | A=R3 |

**PART THREE: LIBRARY OF PROGRAMS**
Again? 
No → end 
One less digit

<table>
<thead>
<tr>
<th>Code</th>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>8AC</td>
<td>?A≠0</td>
<td>A</td>
</tr>
<tr>
<td>60</td>
<td>GOYES</td>
<td>12</td>
</tr>
<tr>
<td>61B0</td>
<td>GOTO</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>CC</td>
<td>A</td>
</tr>
<tr>
<td>103</td>
<td>R3=A</td>
<td></td>
</tr>
<tr>
<td>AC2</td>
<td>C=0</td>
<td>S</td>
</tr>
<tr>
<td>13</td>
<td>111</td>
<td>A</td>
</tr>
<tr>
<td>DE</td>
<td>C=ex</td>
<td>A</td>
</tr>
<tr>
<td>D7</td>
<td>D=C</td>
<td>A</td>
</tr>
<tr>
<td>DE</td>
<td>C=ex</td>
<td>A</td>
</tr>
<tr>
<td>136</td>
<td>CD0ex</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>C=C+A</td>
<td>A</td>
</tr>
<tr>
<td>C2</td>
<td>C=C+A</td>
<td>A</td>
</tr>
<tr>
<td>C8</td>
<td>B=B+A</td>
<td>A</td>
</tr>
<tr>
<td>C8</td>
<td>B=B+A</td>
<td>A</td>
</tr>
<tr>
<td>DD</td>
<td>CBex</td>
<td>A</td>
</tr>
<tr>
<td>136</td>
<td>CD0ex</td>
<td></td>
</tr>
<tr>
<td>B47</td>
<td>D=D+1</td>
<td>S</td>
</tr>
<tr>
<td>110</td>
<td>A=R0</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>D1=A</td>
<td></td>
</tr>
<tr>
<td>AE2</td>
<td>C=0</td>
<td>B</td>
</tr>
<tr>
<td>14</td>
<td>8AF</td>
<td>A</td>
</tr>
<tr>
<td>60</td>
<td>GOYES</td>
<td>15</td>
</tr>
<tr>
<td>6740</td>
<td>GOTO</td>
<td>17</td>
</tr>
<tr>
<td>15</td>
<td>CF</td>
<td>A</td>
</tr>
<tr>
<td>05</td>
<td>SETDEC</td>
<td>B</td>
</tr>
<tr>
<td>AE0</td>
<td>A=0</td>
<td></td>
</tr>
<tr>
<td>15B0</td>
<td>A=DAT1</td>
<td>1</td>
</tr>
<tr>
<td>A62</td>
<td>C=C+A</td>
<td>B</td>
</tr>
<tr>
<td>15A0</td>
<td>A=DAT0</td>
<td>1</td>
</tr>
<tr>
<td>EE</td>
<td>C=A-C</td>
<td>A</td>
</tr>
<tr>
<td>04</td>
<td>SETHEX</td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td>ABex</td>
<td>A</td>
</tr>
<tr>
<td>132</td>
<td>AD0ex</td>
<td></td>
</tr>
<tr>
<td>15C0</td>
<td>DAT0=C</td>
<td>1</td>
</tr>
<tr>
<td>181</td>
<td>D0=D0-</td>
<td>2</td>
</tr>
<tr>
<td>132</td>
<td>AD0ex</td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td>ABex</td>
<td>A</td>
</tr>
<tr>
<td>181</td>
<td>D0=D0-</td>
<td>2</td>
</tr>
<tr>
<td>1C1</td>
<td>D1=D1-</td>
<td>2</td>
</tr>
<tr>
<td>A82</td>
<td>C=0</td>
<td>P</td>
</tr>
<tr>
<td>96A</td>
<td>?C=0</td>
<td>B</td>
</tr>
<tr>
<td>A0</td>
<td>GOYES</td>
<td>16</td>
</tr>
<tr>
<td>3110</td>
<td>LCHEX</td>
<td>01</td>
</tr>
<tr>
<td>6DBF</td>
<td>GOTO</td>
<td>14</td>
</tr>
</tbody>
</table>
PART THREE: LIBRARY OF PROGRAMS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>AE2</td>
<td>C=0</td>
<td>B</td>
</tr>
<tr>
<td>06</td>
<td>6BF</td>
<td>GOTO</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td>96E</td>
<td>?C#0</td>
<td>B</td>
</tr>
<tr>
<td>90</td>
<td>G0YES</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>846</td>
<td>C=C+1</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>6D8F</td>
<td>GOTO</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>161</td>
<td>D0=D0+</td>
<td>2</td>
</tr>
<tr>
<td>136</td>
<td>CD0ex</td>
<td>D0</td>
<td>CBex</td>
</tr>
<tr>
<td>136</td>
<td>CD0ex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>D0=D0+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>847</td>
<td>D=D0+</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>112</td>
<td>A=R2</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>D1=A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1554</td>
<td>DAT1=C</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>171</td>
<td>D1=D1+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>AD1ex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>102</td>
<td>R2=A</td>
<td></td>
</tr>
<tr>
<td>6D9F</td>
<td>GOTO</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>8F2D760</td>
<td>GOSBVL</td>
<td>LOAD_REG</td>
</tr>
<tr>
<td>822</td>
<td>SB=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81943</td>
<td>DSRB</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>832</td>
<td>?SB=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A1</td>
<td>G0YES</td>
<td>110</td>
</tr>
<tr>
<td>174</td>
<td>D1=D1+</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>A=DAT1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>D1=D1+</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>147</td>
<td>C=DAT1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>DAT1=A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1C4</td>
<td>D1=D1-</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>145</td>
<td>DAT1=C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1C4</td>
<td>D1=D1-</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>142</td>
<td>A=DAT0</td>
<td>A</td>
</tr>
<tr>
<td>164</td>
<td>D0=D0+</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>808C</td>
<td>PC=(A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No carry
Carry at end
Yes --> stop
Increment quotient
Loop
Write quotient
Loop
Restore regs.
Need to change order of stack objects?
No --> end
Exchange objects in level 2 and level 3
Return to RPL

end
DIV.LM (# AD61h)

CD20 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 110ED 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)

\[
\begin{align*}
\text{FORMAT} & \quad "0" \text{ SWAP + SWAP FORMAT} \quad "0" \text{ SWAP +} \\
\text{IF} & \quad \text{OVER} \quad "00" = \\
\text{THEN} & \quad \text{DROP2} \quad # \text{305h DOERR} \\
\text{ELSE} & \quad \text{DUP} \text{ NEWOB} \text{ DUP} \text{ 1 OVER SIZE 6 PICK SIZE - 1} + \\
& \quad \text{SUB} \text{ DIV.LM SWAP ROT DROP DUP SIZE DUP 5 ROLL} \\
& \quad \text{SIZE - 1} + \text{SWAP SUB} \\
\end{align*}
\]

MULT.C (# 7E7Ch)

\[
\begin{align*}
\text{DUP2 + ZERO.LM MULT.LM 3 ROLLD DROP2 } & \\
\end{align*}
\]

PREPARE (# 18D6h)

\[
\begin{align*}
\text{FORMAT SWAP FORMAT} & \quad \rightarrow \text{ N1 N2} \\
\text{IF} & \quad \text{N1 SIZE N2 SIZE DUP2 } \\
\text{THEN} & \quad \text{DROP2} \text{ N2 N1} \\
\text{ELSE} & \quad \text{IF} \\
& \quad < \\
\text{THEN} & \quad \text{N1 N2} \\
\end{align*}
\]

Part Three: Library of Programs
ELSE
  N1 N2
  IF
  DUP2 >
  THEN
  SWAP
  END
END
ENCODE SWAP ENCODE DUP2 SIZE SWAP SIZE SWAP
  - 0 CHR
WHILE
  DUP2 SIZE >
REPEAT
  DUP +
END
  I ROT SUB +

DECODE(# A04Dh)
  « DUP DUP + SWAP DECODE.LM DROP FORMAT »

ENCODE(# 19ADh)
  «
  "00000000" 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 (\texttt{# FB90h})
\begin{verbatim}
  \texttt{IF}
  \texttt{FORMAT DUP "0" ==}
  \texttt{THEN}
  \texttt{DROP}
  \texttt{ELSE}
  \texttt{DIV.C SWAP DROP FORMAT}
  \texttt{END}
\end{verbatim}

DIV (\texttt{# 600Ah})
\begin{verbatim}
  \texttt{DIV.C DROP FORMAT}
\end{verbatim}

E (\texttt{# 5A9Eh})
\begin{verbatim}
  \texttt{\textasciitilde STR STR\textasciitilde DUP}
  \texttt{\textasciitilde N}
  \begin{verbatim}
    \texttt{"0"}
    \texttt{WHILE}
    \texttt{N DUP 2 \textasciitilde IP 'N' STO}
    \texttt{REPEAT}
    \texttt{DUP +}
    \texttt{END}
    \texttt{1 ROT SUB +}
  \end{verbatim}
\end{verbatim}

POW (\texttt{# D4DBh})
\begin{verbatim}
  \texttt{\textasciitilde STR STR\textasciitilde}
  \texttt{\textasciitilde N}
  \begin{verbatim}
    \texttt{ENCODE 1 ENCODE}
    \texttt{WHILE}
    \texttt{N DUP 1 DISP 0 \textasciitilde}
    \texttt{REPEAT}
    \texttt{IF}
    \texttt{N 2 \textasciitilde DUP IP 'N' STO FP}
    \texttt{THEN}
    \texttt{OVER MULT.C}
  \end{verbatim}
\end{verbatim}
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 ROLLOD +
  ADD.LM DROP DECODE »
This result was obtained using the programs in CALC, listed previously.
Calculating \( \pi \) has always been a fascinating problem for mathematicians. Today, with computers, it is possible to calculate \( \pi \) 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 \( A\tan \) 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:

\[
\pi = 4 \cdot \left( \sum_{n=0}^{\infty} (-1)^n \cdot \left( \frac{1}{2} \right)^{2n+1} + \left( \frac{1}{5} \right)^{2n+1} + \left( \frac{1}{8} \right)^{2n+1} \right)
\]

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 
062862089896680348253421170679821480863132823066470938446095505822317253 
5940812848117450284102701938521105559644622948954303819644288109756569 
53446128475648233786783165271201909145648566923460346104543266482133936 
072602491412737245870066063155881748881520920962892540917153643678925903 
6001133053054882046652138414695194151160943330572703657595919530921861173 
819326117931051185480744623799627495673518857527248912279381830119491298 
336733624406566430860213949463952247371907021798609437027705392171762931 
76752846748184676694051320005681271452635608277857713427577896091736371 
787214684409012249534301465495853710507922796892589235420199561121290219 
608640344181598136297747713099605187072113499999983729780499510597317328 
16096318595024454955346908302642522308253344685035261931181710100031378 
387528865875332083814206171776691473035982534904287554687311595628638823

### Code

```plaintext
PI (# CF6Dh)

```PI```

\[
\rightarrow P
\]

1 P 1 + E DUP 2 DIV OVER 5 DIV
\]

```
ROT 8 DIV 1 0 0
```

\[
\rightarrow A B C N T S
\]

```
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)
  2 1 2 1 1 2 VAL
  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)
  3 2 1 DER
  ARRAY LIST DROP A
  IF
    A 0 ==
    THEN [ 0 ]
    ELSE 1 A
    FOR X
      X * A ROLLD
    NEXT
  A 1 LIST ARRAY
  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 »
```
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: \[ \text{[ 1 2 1 ] [ 1 1 ] DIVP} \]. The program will return the quotient in level 2, and the remainder in level 1.

**DIVP**

DIVP (# 28E3h)

\[
\text{DUP2 \to A B}
\]

\[
\text{B 1 GET A SIZE 1 GET B SIZE 1 GET DUP2 -} \\
\text{\to c n p q}
\]

\[
\text{IF}\ 
\text{p 1 ==}
\text{THEN}\ 
\text{DROP2 A c / [ 0 ]}
\text{ELSE}\ 
\text{0 q}
\text{FOR x}
\text{OVER 1 GET c / DUP 4 ROLLD * n x -} \\
\text{1 \toLIST RDM}
\text{- ARRY\to 1 GET 1 - \toARRY SWAP DROP B}
\text{NEXT}
\text{DROP q 2 + ROLLD q 1 + \toARRY SWAP}
\]

\]
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 \( x^i: [a_n \ldots a_0] \), 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 \( x^6 - 14x^4 + 49x^2 - 36 \), just type:

\[
[1 \ 0 \ -14 \ 0 \ 49 \ 0 \ -36] \text{ LAGU}
\]

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 " »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 " / 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':

\[
\{ 'SQ(X)+SQ(Y)=1' 'SQ(X-1)+SQ(Y)=1' \} 'MEQ' \text{ STO}
\]

Next, you place the names of the unknowns in a list and store it in 'MVAR'. (In this example: \{ X Y \} 'MVAR' \text{ 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:

\[
\begin{align*}
4: & \quad 3: \\
3: & \quad 2: X: \quad .5 \\
2: & \quad 1: Y: \quad .866025403782
\end{align*}
\]

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 \text{ON}. To stop it, press \text{ON} twice rapidly.

mSOLVER was written by Christophe Dupont de Dinechin.
JACOB (\# E86h)

\[
\begin{align*}
\text{\texttt{\textbackslash{}}} & \text{\texttt{\textbackslash{}}} \text{\texttt{E V}} \\
\text{\texttt{\textbackslash{}}} \text{\texttt{\textbackslash{}}} \text{\texttt{tmp.jacob}} \text{\texttt{\textbackslash{}}} \text{\texttt{CRDIR}} & \\
\text{\texttt{tmp.jacob}} \text{\texttt{\textbackslash{}}} \text{\texttt{CLVAR}} & \\
\{ \} \text{\texttt{1 E SIZE}} & \\
\text{\texttt{FOR e}} & \\
\text{\texttt{1 V SIZE}} & \\
\text{\texttt{FOR v}} & \\
\text{\texttt{E e GET V v GET \texttt{\textbackslash{}}} +} & \\
\text{\texttt{NEXT}} & \\
\text{\texttt{NEXT}} & \\
\text{\texttt{UPDIR \textbackslash{}}} \text{\texttt{tmp.jacob}} \text{\texttt{PURGE}} & \\
\end{align*}
\]

\[
\mu\text{SOLVER} (\# \text{\texttt{CC3Dh}})
\]

\[
\begin{align*}
\text{\texttt{\textbackslash{}}} & \text{\texttt{\textbackslash{}}} \text{\texttt{CLLCD MEQ MVAR \textbackslash{}}} \text{\texttt{P E V}} \\
\text{\texttt{\textbackslash{}}} \text{\texttt{\textbackslash{}}} & \\
\text{\texttt{E V JACOB E SIZE V SIZE}} & \\
\text{\texttt{\textbackslash{}}} & \text{\texttt{J SE SV}} & \\
\text{\texttt{\textbackslash{}}} & \\
\text{\texttt{DO}} & \\
\text{\texttt{1 SV}} & \\
\text{\texttt{FOR v}} & \\
\text{\texttt{V v GET}} & \\
\text{\texttt{IFERR}} & \\
\text{\texttt{RCL}} & \\
\text{\texttt{THEN}} & \\
\text{\texttt{DROP \texttt{\textbackslash{}}} \text{\texttt{\textquote{Variable Error\textquoteright}}} \text{\texttt{1 DISP 0}}} & \\
\text{\texttt{END}} & \\
\text{\texttt{DUP V v GET STO}} & \\
\text{\texttt{NEXT}} & \\
\text{\texttt{SV 1 \texttt{\textbackslash{}}} \text{\texttt{\textbackslash{}}} \text{\texttt{LIST \textbackslash{}}} \text{\texttt{\textbackslash{}}} \text{\texttt{ARRY}}} & \\
\text{\texttt{1 SE}} & \\
\text{\texttt{FOR e}} & \\
\text{\texttt{E e GET}} & \\
\text{\texttt{IFERR}} & \\
\text{\texttt{\texttt{\textbackslash{}}} \text{\texttt{NUM}}} & \\
\text{\texttt{THEN}} & \\
\text{\texttt{\texttt{\textquote{Function Error: \textquoteright}}} \text{\texttt{ERRM} \texttt{\textbackslash{}}} \text{\texttt{+ 1 DISP 0}}} & \\
\text{\texttt{END}} & \\
\end{align*}
\]
```
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 ≤ 1 MVAR SIZE
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).

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:

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Those binary integers would be (ignore the line breaks):

1.  # 10100010100110001010100001010101110001000b
   111010100000010101011110001000b

2.  # 00000000111101000001010101010101010
   0110101011110100111100100010010b

3.  # 011101001000011001111000100101
   011101111000000000011101110001000b

4.  # 001101001000011101100001010111
   0101000101010110010100100100010b

Converting these to hexadecimal, in order, gives the following list:

{ # A298AA2AEA02AE10h # FA0AAA6A09F28Ah
  # 7406784576007788h # 3487305751565482h }
Here is the listing of programs to enter:

\[
\text{AL} (# \text{5B7Ah})
\]
\[
\triangleleft 0 \text{ RDZ 16 RAND } \ast \text{ IP } \triangleright
\]

\[
\text{BL1} (# \text{4998h})
\]
\[
\text{(This is not a NEWLINE character but rather ASCII 127, obtained via 127 CHRS)}
\]

\[
\text{BL2} (# \text{3D27h})
\]
\[
\text{(a single space)}
\]

\[
\text{TS} (# \text{3E54h})
\]
\[
\triangleleft \text{ R} \rightarrow \text{C} 50 \neq \triangleright
\]

\[
\text{TV} (# \text{5A0Dh})
\]
\[
\triangleleft \text{ TVP } \text{ SWAP TVP } + \triangleright
\]

\[
\text{TVP} (# \text{115Fh})
\]
\[
\triangleleft \text{ DUP } 0 \langle \text{ SWAP 15 } \rangle + \triangleright
\]

\[
\text{ETAT} (# \text{85Ah})
\]
\[
\triangleleft 
\begin{align*}
\text{DUP2} \\
\text{IF} \\
\text{TV} \\
\text{THEN} \\
\text{TS} \\
\text{ELSE} \\
\text{DUP2} \ 8 / \text{ IP SWAP 8 / IP 2 } \ast + 1 + \text{ LAB SWAP} \\
\text{GET 3 ROLLD 8 MOD SWAP 8 MOD SWAP 16 SWAP} \\
\text{^ DUP } \# 1h \ast \ast \text{ SWAP 2 SWAP } \ast \ast \text{ AND } \# 0h > \\
\text{END}
\end{align*}
\triangleright
\]

\[
\text{I4} (# \text{AC1h})
\]
\[
\triangleleft X \ 1 \ - \ Y \ \triangleright
\]
I3 (# 8E47h)
  \( \times \ \ 1 + \ Y \) 

I2 (# D48h)
  \( \times \ Y \ \ 1 - \ )

I1 (# E9E3h)
  \( \times \ 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)
  \( \times \ \ ETAT \ 95 \ \ \ast \ 32 \ + \ \ CHR \) 

MAZES (# 38FBh)
  \{ 
  (11,16) 
  \{ \ # \ A298A2A9AA02AE10h \ # \ FA09A6A09F28Ah \\
  \ # \ 7406784576007708h \ # \ 3487305751565482h \ \} 
  \}

Part Three: Library of Programs
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 'SD' 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:

<table>
<thead>
<tr>
<th>Guess No x</th>
</tr>
</thead>
<tbody>
<tr>
<td>8834</td>
</tr>
<tr>
<td>Correct=</td>
</tr>
<tr>
<td>Found=</td>
</tr>
</tbody>
</table>

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(# 2807h)
  «
  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
  »
  »
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
    END
    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
»
The goal of this game is to arrive at a display of the “magic square,” which is the following figure:

```
  1 2 3
  4 5 6
  7 8 9
```

To accomplish this, the player may press different 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
>
```
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 than 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)
< "•" + \rightarrow S
<
  WHILE
  S SIZE
  REPEAT
    S DUP "•" POS DUP2 1 + OVER SIZE
    SUB 'S' STO 1 SWAP 1 - SUB
    \rightarrow T
    <
      WHILE
        T SIZE
        REPEAT
          T 1 40 SUB 1 \^GROB PR1 DROP T 41
          OVER SIZE SUB 'T' STO
        END
    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 \( \rightarrow \text{GROB} \) 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 \texttt{SCREEN}. The lines are added using the OR function on a blank \texttt{GROB}, and then the result is displayed using the \( \rightarrow \text{LCD} \) function.

This program can be used with the program \texttt{DISASM} (the disassembler) to view the listing as it is disassembled. DSP can replace the RPL function \texttt{DISP}. To do this, replace \texttt{1 DISP} in the program \texttt{STOS} with DSP and add \texttt{INITSCR} to the beginning of the program \texttt{DISASM}.

\begin{verbatim}
INITSCR(# 424h)
  « { } 'SCREEN' STO CLLCD »
\end{verbatim}
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 *
  → 0
  
  FOR X
    # 0h 0 # 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)
<
  \rightarrow L
<
  "CCE2D9" \# 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
  "00000000007135147D717414317413706D68A0D08F6A7"
  "1069DF07F2D760142164888C" + + GASS
>

PART THREE: LIBRARY OF PROGRAMS
CCD20 CON(5) PROL_CODE

start

***** CON(5) (end)-(start)

Code length

8FB9760 GOSBVL SAVE_REG

Backup regs.

8E**** GOSUBL 11

LIST OF NOTES—Frequency / Duration (in milliseconds)

CON(5) #00000

CON(5) #00000

I1

07 C=RSTK

135 D1=C

147 C=DAT1 A

D7 D=C A

174 D1=D1+ 5

143 A=DAT1 A

174 D1=D1+ 5

137 CDlex

06 RSTK=C

D6 C=A A

8AA ?C=0 A

D0 GOYES 12

8F6A710 GOSBVL BEEP_LM

69DF GOTO 11

Beep

Loop

I2

07 C=RSTK

8F2D760 GOSBVL LOAD_REG

142 A=DAT0 A

164 D0=D0+ 5

888C PC=(A)

Restore regs.

Return to RPL

End of notes

Read frequency

Read duration

Done?
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 58 .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 CONC(S) PROL CODE Code object
start 15000 CONC(S) (end)-(start) Code length
8FB9760 GOSBV L SAVE_REG Backup regs.
34***** LCHEX START Start frequency
D7 D=C A
DB C=D A
06 RSTK=C (In milliseconds)
34***** LCHEX DUR Beep
8F6A710 GOSBV L BEEP_LM
07 C=RSTK
D7 D=C A
34***** LCHEX INCREMENT Increment
C3 D=D+C A
34***** LCHEX END Ending frequency
*** ?D=C A
7D GOYES 11 Or: ?D>=C A
8F2D760 GOSBV L LOAD_REG Loop
142 A=DAT0 A
164 D0=D0+ A
808C PC=(A) Restore regs.
end Return to RPL
```

316 PART THREE: LIBRARY OF PROGRAMS
\texttt{MODUL(# 1E1Fh)}
\begin{verbatim}
  \rightarrow D F I P
  
  IF
  P 1000 \times DUP 'P' STO \# 0h + \# 0h ==
  THEN
    "ZERO DURATION..." DOERR
  END
  IF
  I \# 0h + \# 0h ==
  THEN
    "ZERO INCREMENT..." DOERR
  END
  "CCD28150088FB9/6034" D A\rightarrow STR + "D7DB0634" +
P A\rightarrow STR + "8F6A71007D734" + I A\rightarrow STR +
IF
  D F <
  THEN
    "C3"
  ELSE
    "E3"
  END
  + "34" + F A\rightarrow STR +
IF
  D F <
  THEN
    "8BF"
  ELSE
    "8BB"
  END
  + "7D8F2D760142164B88C" + GASS
\end{verbatim}
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)

```
\( \begin{align*}
\text{CLLCD} & \text{ # 4E2CFh SYSEVAL RCLF} \\
& \rightarrow \mathbf{F} \\
& \rightarrow \\
& \quad -42 \text{ SF } \{ \} + \text{DATE FP 100 } \times \text{SWAP OVER IP } + \\
& \quad \text{SWAP FP 10000 } + \text{DUP DUP SIZE 2 MOD 2 +} \\
& \quad \text{GET SWAP 1 GET } \\
& \quad \rightarrow \text{Y M} \\
& \rightarrow \\
& \quad 1.0119 \text{ 1 M 100 } / + \text{Y 1000000 } / + \text{DDAYS} \\
& \rightarrow \text{S} \\
& \rightarrow \\
& \quad \{ \text{"JANUARY" "FEBRUARY" "MARCH" "APRIL"} \\
& \quad \text{"MAY" "JUNE" "JULY" "AUGUST"} \\
& \quad \text{"SEPTEMBER" "OCTOBER" "NOVEMBER"} \\
& \quad \text{"DECEMBER"} \\
& \quad \} \\
& \rightarrow \text{M GET } " " \times + \text{Y } + " \\
& 1 22 4 \text{ PICK SIZE } - 2 \text{ / SUB SWAP } + \\
& 1 \text{ DISP } " \text{ MO TU WE TH FR SA SU" 2 DISP} \\
& \{ 31 \text{ 28} 31 \text{ 30} 31 \text{ 30} 31 \text{ 31} 30 \text{ 30}\} 
\end{align*} \)
```

320

PART THREE: LIBRARY OF PROGRAMS
M GET M 2 == Y 4
MOD 0 == Y 100 MOD 0 == - Y 1000 MOD 0 == + AND +
N
0 5
FOR L
" " 1 ?
FOR C
L ? * 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
? 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 C0D20 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
A0000 82190 70000 70534 40200 C9137 1491B 56507
14213 216EA F0142 8A067 70000 7D534 A6200 C9137
14181 C1CD1 41AF0 142CC 81C81 CE4E4 81C1C F1C01
CF141 184AF 01428 A8721 C0141 81C1C D1411 6917F
17F17 41321 4178B 0208F 2D760 14216 4808C 13713
51341 6ED21 53332 0059B E0332 4009B 2A032 003AB
AAB61 7D133 EA131 8000D 21571 6900D 22001 23D1D
71F81 C0013 37F20 1FC90 00133 71201 F6000 01337
3100B E9152 19184 0FA20 01A88 90A00 C8A0E 65FF0
D01D0 10111 0102C 43430 000E2 10811 111AE A2430
F9026 06160 78601 10243 0F906 D1119 C636C A2034
60000 CA100 69201 1AD51 19E9C 6C6CA 2034A 0000C
A1001 1ACE1 0A119 E6109 6FBF1 1111A 8A600 72000
18407 54011 9D511 A109D 910A7 130F9 D910A 76201
19FA1 0AF9D 91097 310F9 D9109 87090 8506A BF201
1A908 24020 00EA1 1CEA3 40400 0BE0 0C434 11000
D7340 3217C F400C 268FF 13711 13414 000CA 11BCA
34380 003BE 00D6A F0DAA E781C 81C13 7C213 5A64A
64AEB B62AE A301A 6C490 A1666 FF153 1E01E 15110
11531 B9C0E 1EB9C 15110 18213 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
4B2A2 CON(5)  #2A2B4
66BC1 CON(5)  #1CB66
76BA1 CON(5)  #1AB67
85BA1 CON(5)  #1A958
CCD20 CON(5)  PROL_CODE
23A00 CON(5)  (end)-(start)
8FB9760 GOSBVL SAVE_REG
1BE0507 D0=(5)  7050E
142 A=DAT0  A
3412000 LCHEX  #00021
C2 C=C+A  A
134 D0=C
10A R2=C
137 CD1ex
135 D1=C
06 RSTK=C
AE0 A=0  B
8082180 LAHEX  #08
100 R0=A
AE0 A=0  B
8082120 LAHEX  #02
101 R1=A
07 C=RSTK
135 D1=C
143 A=DAT1  A
131 D1=A
179 D1=D1+  10
137 CD1ex
135 D1=C
06 RSTK=C
Loop
1BE0507 D0=(5)  7050E
142 A=DAT0  A
130 D0=R
16F D0=D0+  16
16F D0=D0+  16
160 D0=D0+  1
07 C=RSTK
135 D1=C
D0 A=0  A
14B A=DAT1  B
96C ?A##0  B
80 GOYES Cont
8CD990 GOLONG Done

Program object
Null
CHR
Addition
CLLCD
Code object
Code length
Backup regs.
A=@ screen bitmap

Current position

Big pixel height

Big pixel width
(2 nibbles,8 bits)

A=@ string

D1=@ of first char.

D0=@ screen bitmap
ds screen position

D1=@ char.

Read 1 char.
CHR(0)?
Continue
Done
Cont 34F1000  LCHEX  #0001F
EE  C=A-C  A
DA  A=C  A
C6  C=C+C  A
C6  C=C+C  A
C6  C=C+C  A
C6  C=C+C  A
C6  C=C+C  A
C2  C=C+A  A
C2  C=C+A  A
DA  A=C  A
8E4380  GOSUBL  Get_code

;Calculate the offset to find the representation of the char.
;Char= 35 data
;Char between 31 and 90, then:
;offset=(c-31)*35

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:

```
  O  F  F  F  F  O
  F  0  0  0  0  F
  F  0  0  0  0  F
  F  0  0  0  0  F
  F  0  0  0  0  F
  F  0  0  0  0  F

  * The code for A looks like:
    0FFFFFFF00F000F00F000F000FFFFFF...

  * End of character codes, and beginning of BANNER2
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>07</td>
<td>C=RSTK</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>A=A+C</td>
<td></td>
</tr>
<tr>
<td>171</td>
<td>D1=D1+</td>
<td></td>
</tr>
<tr>
<td>137</td>
<td>CD1ex</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>RSTK=C</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>D1=A</td>
<td></td>
</tr>
<tr>
<td>305</td>
<td>LCHEX</td>
<td>#5</td>
</tr>
<tr>
<td>A97</td>
<td>D=C</td>
<td>WP</td>
</tr>
<tr>
<td>A1F</td>
<td>D=D-1</td>
<td>WP</td>
</tr>
<tr>
<td>560</td>
<td>GONC</td>
<td>St_col</td>
</tr>
<tr>
<td>6F70</td>
<td>GOTO</td>
<td>Blank</td>
</tr>
<tr>
<td>3170</td>
<td>LCHEX</td>
<td>#07</td>
</tr>
<tr>
<td>AE5</td>
<td>B=C</td>
<td>B</td>
</tr>
<tr>
<td>A6D</td>
<td>B=B-1</td>
<td>B</td>
</tr>
<tr>
<td>472</td>
<td>GOC</td>
<td>End_col</td>
</tr>
<tr>
<td>1531</td>
<td>A=DAT1</td>
<td>WP</td>
</tr>
<tr>
<td>118</td>
<td>C=R0</td>
<td></td>
</tr>
<tr>
<td>AIE</td>
<td>C=C-1</td>
<td>WP</td>
</tr>
<tr>
<td>431</td>
<td>GOC</td>
<td>End_repH</td>
</tr>
<tr>
<td>1501</td>
<td>DAT0=A</td>
<td>WP</td>
</tr>
<tr>
<td>16F</td>
<td>D0=D0+</td>
<td>16</td>
</tr>
<tr>
<td>16F</td>
<td>D0=D0+</td>
<td>16</td>
</tr>
<tr>
<td>161</td>
<td>D0=D0+</td>
<td>2</td>
</tr>
<tr>
<td>6CEF</td>
<td>GOTO</td>
<td>Repeat_H</td>
</tr>
<tr>
<td>170</td>
<td>D1=D1+</td>
<td></td>
</tr>
<tr>
<td>68DF</td>
<td>GOTO</td>
<td>Wr_col</td>
</tr>
<tr>
<td>119</td>
<td>C=R1</td>
<td></td>
</tr>
<tr>
<td>AE5</td>
<td>B=C</td>
<td></td>
</tr>
<tr>
<td>1BE0507</td>
<td>D0=(5)</td>
<td>7050E</td>
</tr>
<tr>
<td>146</td>
<td>C=DAT0</td>
<td>A</td>
</tr>
<tr>
<td>134</td>
<td>D0=C</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>C=R0</td>
<td></td>
</tr>
<tr>
<td>AEA</td>
<td>A=C</td>
<td>B</td>
</tr>
<tr>
<td>A64</td>
<td>A=A+A</td>
<td>B</td>
</tr>
<tr>
<td>A64</td>
<td>A=A+A</td>
<td>B</td>
</tr>
<tr>
<td>A64</td>
<td>A=A+A</td>
<td>B</td>
</tr>
<tr>
<td>B6E</td>
<td>C=A-C</td>
<td>B</td>
</tr>
</tbody>
</table>

**Get_code**

- Get code
- #3
- St_col
- #87
- B
- End_col
- WP

**Next**

- A1F
- D=D-1
- WP
- 560
- GONC
- St_col
- 6F70
- GOTO
- Blank

**St_col**

- 3170
- LCHEX
- #07
- AE5
- B=C

**Wr_col**

- A6D
- B=B-1
- B
- 472
- GOC
- End_col
- 1531
- A=DAT1
- WP
- 118
- C=R0

**Repeat_H**

- AIE
- C=C-1
- WP
- 431
- GOC
- End_repH
- 1501
- DAT0=A
- WP
- 16F
- D0=D0+ 16
- 16F
- D0=D0+ 16
- 161
- D0=D0+ 2
- 6CEF
- GOTO
- Repeat_H

**End_repH**

- 170
- D1=D1+ 1
- 68DF
- GOTO
- Wr_col

**End_col**

- 119
- C=R1
- AE5
- B=C
- 1BE0507
- D0=(5) 7050E
- 146
- C=DAT0 A
- 134
- D0=C
- 118
- C=R0
- AEA
- A=C
- A64
- A=A+A
- A64
- A=A+A
- A64
- A=A+A
- B6E
- C=A-C

**C=@@ of data**

- Add offset
- @next char

**Save**

- 5 columns

**If not done**

- Otherwise --> blank

**7 lines**

**Done**

- Read pixel
- Big pixel height

**Write**

- Go to the next line

**Next big pixel**

- We have written on right of screen:
  - Now we must scroll to the left

- Recalculate the number of lines to scroll.
<table>
<thead>
<tr>
<th>Repeat_L</th>
<th>B=B-1</th>
<th>B</th>
<th>Extension of width</th>
</tr>
</thead>
<tbody>
<tr>
<td>471</td>
<td>GOC</td>
<td></td>
<td>Next_col</td>
</tr>
<tr>
<td>1BE0507</td>
<td>D0=(5)</td>
<td>7050E</td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>A=DAT0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>D0=A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7D50</td>
<td>GOSUB</td>
<td></td>
<td>Left</td>
</tr>
<tr>
<td>68EF</td>
<td>GOTO</td>
<td></td>
<td>Repeat_L</td>
</tr>
<tr>
<td>Next_col</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11A</td>
<td>C=R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>D0=C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6D7F</td>
<td>GOTO</td>
<td></td>
<td>Next</td>
</tr>
<tr>
<td>Blank</td>
<td></td>
<td></td>
<td>Adding space between two characters</td>
</tr>
<tr>
<td>11A</td>
<td>C=R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>D0=C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>C=R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AE9</td>
<td>A=C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A64</td>
<td>A=A+A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A64</td>
<td>A=A+A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A64</td>
<td>A=A+A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B6E</td>
<td>C=A-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AE5</td>
<td>B=C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A90</td>
<td>A=0</td>
<td></td>
<td>WP</td>
</tr>
<tr>
<td>Part</td>
<td>A6E</td>
<td>C=C-1</td>
<td>B</td>
</tr>
<tr>
<td>431</td>
<td>GOC</td>
<td></td>
<td>Leftb1</td>
</tr>
<tr>
<td>1501</td>
<td>DAT0=A</td>
<td></td>
<td>WP</td>
</tr>
<tr>
<td>16F</td>
<td>D0=D0+</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>16F</td>
<td>D0=D0+</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>D0=D0+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6CEF</td>
<td>GOTO</td>
<td></td>
<td>Part</td>
</tr>
<tr>
<td>Leftb1</td>
<td>AE9</td>
<td>C=B</td>
<td></td>
</tr>
<tr>
<td>1BE0507</td>
<td>D0=(5)</td>
<td>7050E</td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>A=DAT0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>D0=A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7600</td>
<td>GOSUB</td>
<td></td>
<td>Left</td>
</tr>
<tr>
<td>8C896F</td>
<td>GOLONG</td>
<td></td>
<td>Loop</td>
</tr>
<tr>
<td>Left</td>
<td>06</td>
<td>RSTK=C</td>
<td>B</td>
</tr>
<tr>
<td>AE4</td>
<td>A=B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AE5</td>
<td>B=C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AE6</td>
<td>C=A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>RSTK=C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Part Three: Library of Programs**

- **Scroll**: Scroll the visible part of the display.
- **C= # of lines**: C = # of lines
- **D0=@ screen bitmap**: 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
5DF  GONC  Wait
07   C=RSTK
01   RTN

Nextlft
2F  =  15
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   =  0
163  D0=D0+  4
68BF  GOTO  Loop_lft

Done  8F2D760  GOSBVL  LOAD_REG
142  A=DAT0  A
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
821BE 100AE 08082 12810 10713 51431 31179 13713
5061B E0507 14213 016F1 6F160 07135 D814B 96C80
8CD99 034F1 000EE DAC6C 6C6C6 C6C2C 2C2DA 8E438

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
```

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 486EA E5A90 A64E4 11501 16F16
F41F1 C6FAE 91BE0 50714 21307 608BC 896F0 6AE4A
E5AE6 06A6D 57107 AE534 B8000 CE5DF 07012 F1521
B9415 0116E 1521B 94150 116E1 42F41 50320 16368
BF8F2 D7601 42164 808C8 DBF1B 2130

330

Part Three: Library of Programs
Appendices
A. Answers to Exercises
1-1. (the left-shifted 0 key)

1-2. (the right-shifted STO key)

2-1. One possible sequence is 5 ENTER 3 ENTER 1 + 9 ENTER 5 - \times +. (With some functions, like +, -, and \times, you don't need to press ENTER after pressing them).

2-2. For example SWAP ROT

2-3. \(\text{COS}((3\times5)-11)/4-1)\) which gives 1 (COS(0)).

3-1. Type \(\text{HOME} \text{EXE} \text{HOME} \text{EXE} \text{HOME} \\text{EXE} \text{HOME} \text{EXE} \text{HOME} \text{MEMORY} \text{CINT} \text{VAR} \text{EXE} \text{1} \text{A STO} \text{2} \text{B STO} \text{3} \text{C STO}\)

3-2. 6 (PARTS, PROB, HYP, MATR, VECTR, and BASE).

4-1. \(\langle \rightarrow \text{ A B } \langle \rightarrow \text{ A B } + \rangle \rangle\) 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:
\(\text{FIBO} (\# \text{ 5B7Eh})\)
\(\langle \rightarrow \text{ N} \langle \rightarrow \text{ N} \rightarrow \text{ IF} \text{ N} \text{ 1} \rightarrow \text{ THEN} \text{ 1} \rightarrow \text{ ELSE} \text{ N} \text{ 1 - FIBO N} \text{ 2 - FIBO} +\rightarrow \text{ END}\rightarrow \rangle\)

5-1. 1h, Ah, 19h, FFFFh, BEBEh.

5-2. 291, 16, 256, 2898, 3.

A. Answers to Exercises
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
65000 beginCON(5) (end)-(begin)
6310 GOTO 11
CC sub1 A=A-1 A
3454321 LCHEX #12345
CE l2 C=C-1 A
5DF GONC 12
03 RTNCC
3450000 l1 LCHEX #0005
DA A=C A
?CEF l3 GOSUB 12
8AC ?A#0 A
9F GOYES 13
3410000 LCHEX #0001
DA A=C A
7110 GOSUB 14
8A8 ?A=0 A
40 GOYES 15
CC A=A-1 A
142 l5 A=DAT0 A
164 D0=D0+ 5
808C PC=(A)
```
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
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
888C   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 0000000001 0 10000 000000002 0

A. Answers to Exercises
8-9. \((-33, 33)\).
8-10. D9920 0000 00000000000000 0 0000 ...
    ...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 1BF793500000000
8-19. #54321h
8-20. 47A20B2130
8-21. { OK }
8-22. 69A20 FF7 12000 00000 10 44 10 C2A207000033
    21000 10 14 10 C2A20700043
8-23. 69A20 100 321 03EF7 00000 12000 00000 10 44 10
    C2A207000033 21000 10 14 10 C2A20700043
8-24. 8BA20 84E201014 84E201024 76BA1 B2130
8-25. 'A*(B-C)'
8-26. ADA20 339200000000000000210 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
8-32. '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.

---

A. Answers to Exercises
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 affect 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 difference being the symbol list used.

For example, to count in base 4, we would have: 0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 100, 101,... (which, in base 10, corresponds to the sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,...).

Note, however, that the number 102, 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, 11, 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, 0011, 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, ..., 11111110, 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)
  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
```

B. Background Information
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 #1RA1Fh 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Library Code</th>
<th>Memory Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>#630h</td>
<td>61</td>
<td>19A1Fh</td>
</tr>
<tr>
<td>ACK</td>
<td>#815h</td>
<td>21</td>
<td>19B7Eh</td>
</tr>
<tr>
<td>ACKALL</td>
<td>#814h</td>
<td>28</td>
<td>19B63h</td>
</tr>
<tr>
<td>ACOS</td>
<td>#658h</td>
<td>88</td>
<td>1872Fh</td>
</tr>
<tr>
<td>ACOSH</td>
<td>#658h</td>
<td>91</td>
<td>18B3Dh</td>
</tr>
<tr>
<td>ALOG</td>
<td>#868h</td>
<td>96</td>
<td>18B3Dh</td>
</tr>
<tr>
<td>AND</td>
<td>#865h</td>
<td>229</td>
<td>17E83h</td>
</tr>
<tr>
<td>APPLY</td>
<td>#182h</td>
<td>258</td>
<td>1F550h</td>
</tr>
<tr>
<td>APPLY</td>
<td>#183h</td>
<td>259</td>
<td>1F55Ch</td>
</tr>
<tr>
<td>ARC</td>
<td>#908h</td>
<td>216</td>
<td>1E502h</td>
</tr>
<tr>
<td>ARCHIVE</td>
<td>#160h</td>
<td>352</td>
<td>125Ah</td>
</tr>
<tr>
<td>ARG</td>
<td>#840h</td>
<td>77</td>
<td>1E20Bh</td>
</tr>
<tr>
<td>ARRY+</td>
<td>#90Ah</td>
<td>171</td>
<td>10092h</td>
</tr>
<tr>
<td>+ARRY</td>
<td>#90Ah</td>
<td>170</td>
<td>10093h</td>
</tr>
<tr>
<td>ASIN</td>
<td>#958h</td>
<td>87</td>
<td>186Ah</td>
</tr>
<tr>
<td>ASINH</td>
<td>#95Ah</td>
<td>98</td>
<td>187E8h</td>
</tr>
<tr>
<td>ASINH</td>
<td>#95Ah</td>
<td>98</td>
<td>187E8h</td>
</tr>
<tr>
<td>ASINH</td>
<td>#95Ah</td>
<td>80</td>
<td>19576h</td>
</tr>
<tr>
<td>ATAN</td>
<td>#959h</td>
<td>89</td>
<td>1879Ch</td>
</tr>
<tr>
<td>ATANH</td>
<td>#95Ch</td>
<td>92</td>
<td>1882eh</td>
</tr>
<tr>
<td>ATANH</td>
<td>#95Ch</td>
<td>92</td>
<td>1882eh</td>
</tr>
<tr>
<td>ATANH</td>
<td>#95Ch</td>
<td>92</td>
<td>1882eh</td>
</tr>
<tr>
<td>ATTACH</td>
<td>#165h</td>
<td>357</td>
<td>21448h</td>
</tr>
<tr>
<td>AUTO</td>
<td>#83Ch</td>
<td>192</td>
<td>1E1ABh</td>
</tr>
<tr>
<td>AXES</td>
<td>#88Ah</td>
<td>186</td>
<td>1E86Eh</td>
</tr>
<tr>
<td>BAR</td>
<td>#8E3h</td>
<td>227</td>
<td>1E741h</td>
</tr>
<tr>
<td>BARPLOT</td>
<td>#13Ch</td>
<td>316</td>
<td>28133h</td>
</tr>
<tr>
<td>BINS</td>
<td>#172h</td>
<td>378</td>
<td>2286Ch</td>
</tr>
<tr>
<td>B+R</td>
<td>#90Ah</td>
<td>10</td>
<td>1968Bh</td>
</tr>
<tr>
<td>BEEP</td>
<td>#934h</td>
<td>52</td>
<td>1A5C4h</td>
</tr>
<tr>
<td>BESTFIT</td>
<td>#143h</td>
<td>323</td>
<td>2825Es</td>
</tr>
<tr>
<td>BIN</td>
<td>#998h</td>
<td>144</td>
<td>1C559h</td>
</tr>
<tr>
<td>BINS</td>
<td>#138h</td>
<td>315</td>
<td>28186h</td>
</tr>
<tr>
<td>BLANK</td>
<td>#801h</td>
<td>289</td>
<td>1E416h</td>
</tr>
<tr>
<td>BOX</td>
<td>#908h</td>
<td>288</td>
<td>1E9ECh</td>
</tr>
<tr>
<td>BUFLEN</td>
<td>#175h</td>
<td>374</td>
<td>22208h</td>
</tr>
<tr>
<td>BYTES</td>
<td>#925h</td>
<td>38</td>
<td>1A19Dh</td>
</tr>
<tr>
<td>C+PX</td>
<td>#8C7h</td>
<td>199</td>
<td>1E299h</td>
</tr>
<tr>
<td>C+R</td>
<td>#89Fh</td>
<td>159</td>
<td>1C9Eh</td>
</tr>
<tr>
<td>CEIL</td>
<td>#968h</td>
<td>104</td>
<td>1B8CFh</td>
</tr>
<tr>
<td>CENTR</td>
<td>#889h</td>
<td>187</td>
<td>1E8EBh</td>
</tr>
<tr>
<td>CF</td>
<td>#989h</td>
<td>132</td>
<td>1C205h</td>
</tr>
<tr>
<td>%CH</td>
<td>#87Eh</td>
<td>126</td>
<td>1C149h</td>
</tr>
<tr>
<td>CHR</td>
<td>#905h</td>
<td>165</td>
<td>1C966h</td>
</tr>
<tr>
<td>CKSM</td>
<td>#171h</td>
<td>369</td>
<td>21FECh</td>
</tr>
<tr>
<td>CLEAR</td>
<td>#11Ah</td>
<td>282</td>
<td>1FCEBH</td>
</tr>
<tr>
<td>CLKADJ</td>
<td>#818h</td>
<td>24</td>
<td>19B0Eh</td>
</tr>
<tr>
<td>CLMED</td>
<td>#938h</td>
<td>56</td>
<td>1A959h</td>
</tr>
<tr>
<td>CLOSEIO</td>
<td>#16Ah</td>
<td>362</td>
<td>21E5Dh</td>
</tr>
<tr>
<td>CLE</td>
<td>#11Ch</td>
<td>284</td>
<td>1FD29h</td>
</tr>
<tr>
<td>CLUSR</td>
<td>#15Ah</td>
<td>346</td>
<td>210FCh</td>
</tr>
<tr>
<td>CLVAR</td>
<td>#15Ah</td>
<td>346</td>
<td>210FCh</td>
</tr>
<tr>
<td>CNRM</td>
<td>#077h</td>
<td>119</td>
<td>1BF8Eh</td>
</tr>
<tr>
<td>COLCT</td>
<td>#104h</td>
<td>333</td>
<td>12B1Ah</td>
</tr>
<tr>
<td>COLE</td>
<td>#136h</td>
<td>312</td>
<td>1289Ah</td>
</tr>
<tr>
<td>COMB</td>
<td>#081h</td>
<td>129</td>
<td>11C16h</td>
</tr>
<tr>
<td>CON</td>
<td>#0ADh</td>
<td>173</td>
<td>1D186h</td>
</tr>
<tr>
<td>CONIC</td>
<td>#0CDh</td>
<td>221</td>
<td>1E681h</td>
</tr>
<tr>
<td>CONJ</td>
<td>#136h</td>
<td>62</td>
<td>1A96Eh</td>
</tr>
<tr>
<td>CONT</td>
<td>#83Ah</td>
<td>58</td>
<td>1A868h</td>
</tr>
<tr>
<td>CONVERT</td>
<td>#088h</td>
<td>11</td>
<td>160B6h</td>
</tr>
<tr>
<td>CORR</td>
<td>#121h</td>
<td>269</td>
<td>1FDC1h</td>
</tr>
<tr>
<td>COS</td>
<td>#052h</td>
<td>82</td>
<td>1B565h</td>
</tr>
<tr>
<td>COSH</td>
<td>#055h</td>
<td>85</td>
<td>1B686h</td>
</tr>
<tr>
<td>COV</td>
<td>#122h</td>
<td>298</td>
<td>1FDC0h</td>
</tr>
<tr>
<td>CR</td>
<td>#0F3h</td>
<td>243</td>
<td>1EEA4h</td>
</tr>
<tr>
<td>CROIR</td>
<td>#029h</td>
<td>32</td>
<td>1A185h</td>
</tr>
<tr>
<td>CROSS</td>
<td>#079h</td>
<td>122</td>
<td>1C86Eh</td>
</tr>
<tr>
<td>DATE</td>
<td>#011h</td>
<td>17</td>
<td>19812h</td>
</tr>
<tr>
<td>+DATE</td>
<td>#016h</td>
<td>22</td>
<td>19896h</td>
</tr>
<tr>
<td>DATE+</td>
<td>#01Fh</td>
<td>31</td>
<td>19902h</td>
</tr>
<tr>
<td>D+R</td>
<td>#079h</td>
<td>112</td>
<td>1BE1Ch</td>
</tr>
<tr>
<td>DORAYS</td>
<td>#01Ch</td>
<td>30</td>
<td>19962h</td>
</tr>
<tr>
<td>DEC</td>
<td>#091h</td>
<td>145</td>
<td>1C574h</td>
</tr>
<tr>
<td>DECR</td>
<td>#14Ch</td>
<td>392</td>
<td>289Ah</td>
</tr>
<tr>
<td>DEFINE</td>
<td>#156h</td>
<td>342</td>
<td>20656h</td>
</tr>
<tr>
<td>DEG</td>
<td>#087h</td>
<td>135</td>
<td>1C399h</td>
</tr>
<tr>
<td>DELALARM</td>
<td>#01Ch</td>
<td>28</td>
<td>19972h</td>
</tr>
<tr>
<td>DELAY</td>
<td>#0F5h</td>
<td>245</td>
<td>1E4F3h</td>
</tr>
<tr>
<td>DELKEYS</td>
<td>#170h</td>
<td>381</td>
<td>22548h</td>
</tr>
<tr>
<td>DEPND</td>
<td>#04Ch</td>
<td>196</td>
<td>1E228h</td>
</tr>
<tr>
<td>DEPTH</td>
<td>#114h</td>
<td>276</td>
<td>1F444h</td>
</tr>
<tr>
<td>DET</td>
<td>#076h</td>
<td>120</td>
<td>1B0DEh</td>
</tr>
<tr>
<td>DETACH</td>
<td>#166h</td>
<td>358</td>
<td>2147Ch</td>
</tr>
<tr>
<td>DISP</td>
<td>#032h</td>
<td>58</td>
<td>1A594h</td>
</tr>
<tr>
<td>DOERR</td>
<td>#029h</td>
<td>42</td>
<td>1A339h</td>
</tr>
<tr>
<td>DOT</td>
<td>#079h</td>
<td>121</td>
<td>1BBFEh</td>
</tr>
<tr>
<td>DRAW</td>
<td>#08Fh</td>
<td>191</td>
<td>1E190h</td>
</tr>
<tr>
<td>DRAX</td>
<td>#0Clh</td>
<td>193</td>
<td>1E1CBh</td>
</tr>
<tr>
<td>DROP</td>
<td>#119h</td>
<td>272</td>
<td>1F808h</td>
</tr>
<tr>
<td>DROP2</td>
<td>#111h</td>
<td>273</td>
<td>1BF3Fh</td>
</tr>
<tr>
<td>DROPN</td>
<td>#115h</td>
<td>277</td>
<td>1FC64h</td>
</tr>
<tr>
<td>DTMG</td>
<td>#180h</td>
<td>304</td>
<td>22633h</td>
</tr>
<tr>
<td>DUP</td>
<td>#180h</td>
<td>269</td>
<td>1F88Dh</td>
</tr>
<tr>
<td>DUP2</td>
<td>#180h</td>
<td>278</td>
<td>1F8A2h</td>
</tr>
<tr>
<td>DUPN</td>
<td>#116h</td>
<td>279</td>
<td>1FC7Fh</td>
</tr>
<tr>
<td>ENG</td>
<td>#08Ch</td>
<td>148</td>
<td>1C452h</td>
</tr>
<tr>
<td>EQ+</td>
<td>#096h</td>
<td>168</td>
<td>1CE6Eh</td>
</tr>
<tr>
<td>ERASE</td>
<td>#0C5h</td>
<td>197</td>
<td>1E29Fh</td>
</tr>
<tr>
<td>ERR0</td>
<td>#028h</td>
<td>43</td>
<td>1A860h</td>
</tr>
<tr>
<td>ERR1</td>
<td>#020h</td>
<td>45</td>
<td>1A8A9h</td>
</tr>
<tr>
<td>ERR2</td>
<td>#022h</td>
<td>44</td>
<td>1A868h</td>
</tr>
<tr>
<td>EVAL</td>
<td>#022h</td>
<td>46</td>
<td>1A88Eh</td>
</tr>
<tr>
<td>EXP</td>
<td>#050h</td>
<td>93</td>
<td>1B905h</td>
</tr>
<tr>
<td>Command</td>
<td>Address</td>
<td>Value 1</td>
<td>Value 2</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>EXPAN</td>
<td>#D14Ch</td>
<td>334</td>
<td>#20A49h</td>
</tr>
<tr>
<td>EXPFIT</td>
<td>#D11Ch</td>
<td>321</td>
<td>#281FBh</td>
</tr>
<tr>
<td>EXPM</td>
<td>#D02h</td>
<td>98</td>
<td>#19AC2h</td>
</tr>
<tr>
<td>e</td>
<td>#D04h</td>
<td>66</td>
<td>#1AB23h</td>
</tr>
<tr>
<td>FACT</td>
<td>#D06h</td>
<td>100</td>
<td>#1B841h</td>
</tr>
<tr>
<td>FC?</td>
<td>#D06h</td>
<td>134</td>
<td>#1C368h</td>
</tr>
<tr>
<td>FCRC</td>
<td>#D06h</td>
<td>143</td>
<td>#1C528h</td>
</tr>
<tr>
<td>FINDALARM</td>
<td>#D08h</td>
<td>27</td>
<td>#19948h</td>
</tr>
<tr>
<td>FINISH</td>
<td>#D16h</td>
<td>367</td>
<td>#21FB6h</td>
</tr>
<tr>
<td>FIX</td>
<td>#D08h</td>
<td>138</td>
<td>#1C3E9h</td>
</tr>
<tr>
<td>FLOOR</td>
<td>#D07h</td>
<td>189</td>
<td>#188D9h</td>
</tr>
<tr>
<td>FP</td>
<td>#D06h</td>
<td>182</td>
<td>#18FB8h</td>
</tr>
<tr>
<td>FREE</td>
<td>#D13h</td>
<td>355</td>
<td>#21301h</td>
</tr>
<tr>
<td>FREEZE</td>
<td>#D33h</td>
<td>51</td>
<td>#1854Ah</td>
</tr>
<tr>
<td>FS?</td>
<td>#D85h</td>
<td>133</td>
<td>#1C313h</td>
</tr>
<tr>
<td>FSIC</td>
<td>#D81h</td>
<td>142</td>
<td>#1C414h</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>#D0Ch</td>
<td>220</td>
<td>#1E661h</td>
</tr>
<tr>
<td>GET</td>
<td>#D02h</td>
<td>178</td>
<td>#107C6h</td>
</tr>
<tr>
<td>GETI</td>
<td>#D03h</td>
<td>179</td>
<td>#10B27h</td>
</tr>
<tr>
<td>GOR</td>
<td>#D03h</td>
<td>211</td>
<td>#1E456h</td>
</tr>
<tr>
<td>GRAD</td>
<td>#D07h</td>
<td>137</td>
<td>#1C3C9h</td>
</tr>
<tr>
<td>GRAPH</td>
<td>#D08h</td>
<td>200</td>
<td>#1E29Ah</td>
</tr>
<tr>
<td>GRDB</td>
<td>#D07h</td>
<td>215</td>
<td>#1E5ADh</td>
</tr>
<tr>
<td>GXR</td>
<td>#D04h</td>
<td>212</td>
<td>#1E4E4h</td>
</tr>
<tr>
<td>H</td>
<td>#D00h</td>
<td>189</td>
<td>#1E158h</td>
</tr>
<tr>
<td>HISTOGRAM</td>
<td>#D02h</td>
<td>146</td>
<td>#1C5F9h</td>
</tr>
<tr>
<td>HISTPLOT</td>
<td>#D02h</td>
<td>226</td>
<td>#1E721h</td>
</tr>
<tr>
<td>HMS</td>
<td>#D04h</td>
<td>317</td>
<td>#20167h</td>
</tr>
<tr>
<td>HMS-</td>
<td>#D07h</td>
<td>16</td>
<td>#1B5E5h</td>
</tr>
<tr>
<td>HMS+</td>
<td>#D07h</td>
<td>117</td>
<td>#1BF7Eh</td>
</tr>
<tr>
<td>HMS-</td>
<td>#D07h</td>
<td>115</td>
<td>#1BF3Eh</td>
</tr>
<tr>
<td>HMS+</td>
<td>#D07h</td>
<td>114</td>
<td>#1BF1Eh</td>
</tr>
<tr>
<td>HOME</td>
<td>#D02h</td>
<td>34</td>
<td>#1A148h</td>
</tr>
<tr>
<td>ION</td>
<td>#D04h</td>
<td>174</td>
<td>#1020Ch</td>
</tr>
<tr>
<td>IFT</td>
<td>#D06h</td>
<td>48</td>
<td>#1A4Cd</td>
</tr>
<tr>
<td>IFTE</td>
<td>#D02h</td>
<td>47</td>
<td>#19F5Eh</td>
</tr>
<tr>
<td>IM</td>
<td>#D09h</td>
<td>155</td>
<td>#1CB19h</td>
</tr>
<tr>
<td>INCR</td>
<td>#D0Bh</td>
<td>391</td>
<td>#20844h</td>
</tr>
<tr>
<td>INDEP</td>
<td>#D07h</td>
<td>193</td>
<td>#1EBA4h</td>
</tr>
<tr>
<td>INPUT</td>
<td>#D17h</td>
<td>378</td>
<td>#2240Ah</td>
</tr>
<tr>
<td>INV</td>
<td>#D04h</td>
<td>76</td>
<td>#1B278h</td>
</tr>
<tr>
<td>IP</td>
<td>#D05h</td>
<td>181</td>
<td>#1B360h</td>
</tr>
<tr>
<td>ISOL</td>
<td>#D15h</td>
<td>336</td>
<td>#20A93h</td>
</tr>
<tr>
<td>i</td>
<td>#D03h</td>
<td>67</td>
<td>#1A845h</td>
</tr>
<tr>
<td>KERRM</td>
<td>#D17h</td>
<td>373</td>
<td>#2206Ch</td>
</tr>
<tr>
<td>KEY</td>
<td>#D09h</td>
<td>57</td>
<td>#1A873h</td>
</tr>
<tr>
<td>KGET</td>
<td>#D1Ch</td>
<td>364</td>
<td>#21F24h</td>
</tr>
<tr>
<td>KILL</td>
<td>#D08h</td>
<td>48</td>
<td>#1A963h</td>
</tr>
<tr>
<td>LABEL</td>
<td>#D05h</td>
<td>281</td>
<td>#1E205h</td>
</tr>
<tr>
<td>LAST</td>
<td>#D06h</td>
<td>54</td>
<td>#1A684h</td>
</tr>
<tr>
<td>LASTARG</td>
<td>#D06h</td>
<td>54</td>
<td>#1A684h</td>
</tr>
<tr>
<td>LCD+</td>
<td>#D05h</td>
<td>213</td>
<td>#1E572h</td>
</tr>
<tr>
<td>*LCD</td>
<td>#D06h</td>
<td>214</td>
<td>#1E580h</td>
</tr>
<tr>
<td>LIBS</td>
<td>#D16h</td>
<td>356</td>
<td>#2142Dh</td>
</tr>
<tr>
<td>LINE</td>
<td>#D0Ch</td>
<td>206</td>
<td>#1E939h</td>
</tr>
<tr>
<td>LINFIT</td>
<td>#D13h</td>
<td>314</td>
<td>#208F3h</td>
</tr>
<tr>
<td>LIST</td>
<td>#D13h</td>
<td>314</td>
<td>#208F3h</td>
</tr>
<tr>
<td>LIST+</td>
<td>#D09h</td>
<td>158</td>
<td>#1C95Ah</td>
</tr>
<tr>
<td>MANT</td>
<td>#D09h</td>
<td>152</td>
<td>#1C783h</td>
</tr>
<tr>
<td>LOG</td>
<td>#D05h</td>
<td>94</td>
<td>#1B94Fh</td>
</tr>
<tr>
<td>LOGFIT</td>
<td>#D16h</td>
<td>320</td>
<td>#201D6h</td>
</tr>
<tr>
<td>LR</td>
<td>#D12h</td>
<td>302</td>
<td>#1FF28h</td>
</tr>
<tr>
<td>MATCH</td>
<td>#D09h</td>
<td>265</td>
<td>#1F599h</td>
</tr>
<tr>
<td>MATCH+</td>
<td>#D10h</td>
<td>266</td>
<td>#1F9ADh</td>
</tr>
<tr>
<td>MAX</td>
<td>#D0Ah</td>
<td>186</td>
<td>#1BC71h</td>
</tr>
<tr>
<td>MAKE</td>
<td>#D12h</td>
<td>296</td>
<td>#1F3E7h</td>
</tr>
<tr>
<td>MAKR</td>
<td>#D10h</td>
<td>64</td>
<td>#1AADFh</td>
</tr>
<tr>
<td>MEAN</td>
<td>#D19h</td>
<td>297</td>
<td>#1FE99h</td>
</tr>
<tr>
<td>MEM</td>
<td>#D15h</td>
<td>344</td>
<td>#20F4Ah</td>
</tr>
<tr>
<td>MENU</td>
<td>#D15h</td>
<td>348</td>
<td>#21196h</td>
</tr>
<tr>
<td>MERGE</td>
<td>#D16h</td>
<td>354</td>
<td>#2137Fh</td>
</tr>
<tr>
<td>MIN</td>
<td>#D09h</td>
<td>187</td>
<td>#1BC3Eh</td>
</tr>
<tr>
<td>MINE</td>
<td>#D12h</td>
<td>298</td>
<td>#1FEB4h</td>
</tr>
<tr>
<td>MNR</td>
<td>#D04h</td>
<td>65</td>
<td>#1A0B1h</td>
</tr>
<tr>
<td>MODE</td>
<td>#D06h</td>
<td>110</td>
<td>#1B4E0h</td>
</tr>
<tr>
<td>NEG</td>
<td>#D03h</td>
<td>68</td>
<td>#1A999h</td>
</tr>
<tr>
<td>NEWOB</td>
<td>#D02h</td>
<td>39</td>
<td>#1A28Ch</td>
</tr>
<tr>
<td>NOT</td>
<td>#D07h</td>
<td>231</td>
<td>#1E88Fh</td>
</tr>
<tr>
<td>NE</td>
<td>#D12h</td>
<td>288</td>
<td>#1F0A6h</td>
</tr>
<tr>
<td>NUM</td>
<td>#D04h</td>
<td>164</td>
<td>#1CB46h</td>
</tr>
<tr>
<td>NUM+</td>
<td>#D05h</td>
<td>53</td>
<td>#1A5E4h</td>
</tr>
<tr>
<td>OBJ+</td>
<td>#D04h</td>
<td>169</td>
<td>#1CF78h</td>
</tr>
<tr>
<td>OCT</td>
<td>#D09h</td>
<td>147</td>
<td>#1C59Ah</td>
</tr>
<tr>
<td>OFF</td>
<td>#D09h</td>
<td>41</td>
<td>#1A31Eh</td>
</tr>
<tr>
<td>OLDPRT</td>
<td>#D0Eh</td>
<td>239</td>
<td>#1EE38h</td>
</tr>
<tr>
<td>OPENIO</td>
<td>#D19h</td>
<td>361</td>
<td>#21E55h</td>
</tr>
<tr>
<td>OR</td>
<td>#D06h</td>
<td>230</td>
<td>#1E889h</td>
</tr>
<tr>
<td>ORDER</td>
<td>#D15h</td>
<td>345</td>
<td>#20FD9h</td>
</tr>
<tr>
<td>ORDER+</td>
<td>#D11h</td>
<td>275</td>
<td>#1FC29h</td>
</tr>
<tr>
<td>PARAMETRIC</td>
<td>#D0Fh</td>
<td>223</td>
<td>#1E6C1h</td>
</tr>
<tr>
<td>PARITY</td>
<td>#D13h</td>
<td>371</td>
<td>#2202Ch</td>
</tr>
<tr>
<td>PATH</td>
<td>#D01h</td>
<td>33</td>
<td>#1A125h</td>
</tr>
<tr>
<td>POIM</td>
<td>#D0Ch</td>
<td>195</td>
<td>#1E281h</td>
</tr>
<tr>
<td>PERM</td>
<td>#D08h</td>
<td>130</td>
<td>#1C236h</td>
</tr>
<tr>
<td>PGDIR</td>
<td>#D15h</td>
<td>351</td>
<td>#2123Ah</td>
</tr>
<tr>
<td>PICK</td>
<td>#D11h</td>
<td>279</td>
<td>#1FC9Ah</td>
</tr>
<tr>
<td>PICT</td>
<td>#D02h</td>
<td>210</td>
<td>#1E436h</td>
</tr>
<tr>
<td>PIX</td>
<td>#D0Ch</td>
<td>265</td>
<td>#1E36Eh</td>
</tr>
<tr>
<td>PIXOFF</td>
<td>#D0Ch</td>
<td>284</td>
<td>#1E344h</td>
</tr>
<tr>
<td>PIXON</td>
<td>#D03h</td>
<td>203</td>
<td>#1E31Ah</td>
</tr>
<tr>
<td>PKT</td>
<td>#D17h</td>
<td>377</td>
<td>#2200Dh</td>
</tr>
<tr>
<td>PMAX</td>
<td>#D09h</td>
<td>185</td>
<td>#1E89Eh</td>
</tr>
<tr>
<td>PMIN</td>
<td>#D08h</td>
<td>184</td>
<td>#1E87Eh</td>
</tr>
</tbody>
</table>
C. RPL Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+TIME</td>
<td>#017h</td>
<td>23</td>
</tr>
<tr>
<td>TLINE</td>
<td>#0CFh</td>
<td>207</td>
</tr>
<tr>
<td>THENU</td>
<td>#15Bh</td>
<td>347</td>
</tr>
<tr>
<td>TOT</td>
<td>#12Ch</td>
<td>300</td>
</tr>
<tr>
<td>TRANSIO</td>
<td>#174h</td>
<td>372</td>
</tr>
<tr>
<td>TRN</td>
<td>#0AFh</td>
<td>175</td>
</tr>
<tr>
<td>TRNC</td>
<td>#06Dh</td>
<td>189</td>
</tr>
<tr>
<td>TRUTH</td>
<td>#0E8h</td>
<td>224</td>
</tr>
<tr>
<td>TSTR</td>
<td>#01Dh</td>
<td>29</td>
</tr>
<tr>
<td>TVARS</td>
<td>#025h</td>
<td>37</td>
</tr>
<tr>
<td>TYPE</td>
<td>#0A6h</td>
<td>166</td>
</tr>
<tr>
<td>UBASE</td>
<td>#08Eh</td>
<td>14</td>
</tr>
<tr>
<td>UFAC</td>
<td>#00Fh</td>
<td>15</td>
</tr>
<tr>
<td>+UNIT</td>
<td>#080h</td>
<td>13</td>
</tr>
<tr>
<td>UPOIR</td>
<td>#023h</td>
<td>35</td>
</tr>
<tr>
<td>UTPC</td>
<td>#134h</td>
<td>388</td>
</tr>
<tr>
<td>UTPF</td>
<td>#136h</td>
<td>318</td>
</tr>
<tr>
<td>UTPN</td>
<td>#135h</td>
<td>389</td>
</tr>
<tr>
<td>UTPT</td>
<td>#137h</td>
<td>311</td>
</tr>
<tr>
<td>UVAR</td>
<td>#08Ch</td>
<td>12</td>
</tr>
<tr>
<td>VARS</td>
<td>#120h</td>
<td>381</td>
</tr>
<tr>
<td>V+</td>
<td>#084h</td>
<td>180</td>
</tr>
<tr>
<td>V2</td>
<td>#085h</td>
<td>181</td>
</tr>
<tr>
<td>V3</td>
<td>#086h</td>
<td>182</td>
</tr>
<tr>
<td>VTYPE</td>
<td>#087h</td>
<td>167</td>
</tr>
<tr>
<td>W</td>
<td>#08Eh</td>
<td>190</td>
</tr>
<tr>
<td>WAIT</td>
<td>#087h</td>
<td>55</td>
</tr>
<tr>
<td>WLOG</td>
<td>#019h</td>
<td>19</td>
</tr>
<tr>
<td>X</td>
<td>#123h</td>
<td>291</td>
</tr>
<tr>
<td>X^2</td>
<td>#125h</td>
<td>293</td>
</tr>
<tr>
<td>XCOL</td>
<td>#132h</td>
<td>386</td>
</tr>
<tr>
<td>XMIT</td>
<td>#167h</td>
<td>359</td>
</tr>
<tr>
<td>XOR</td>
<td>#08Bh</td>
<td>232</td>
</tr>
<tr>
<td>XPON</td>
<td>#069h</td>
<td>105</td>
</tr>
<tr>
<td>XRNG</td>
<td>#0DAh</td>
<td>218</td>
</tr>
<tr>
<td>XROOT</td>
<td>#04Ah</td>
<td>74</td>
</tr>
<tr>
<td>X+Y</td>
<td>#127h</td>
<td>255</td>
</tr>
<tr>
<td>SY</td>
<td>#124h</td>
<td>292</td>
</tr>
<tr>
<td>SY^2</td>
<td>#126h</td>
<td>294</td>
</tr>
<tr>
<td>YCOL</td>
<td>#133h</td>
<td>307</td>
</tr>
<tr>
<td>YRING</td>
<td>#08Oh</td>
<td>219</td>
</tr>
<tr>
<td>+</td>
<td>#044h</td>
<td>68</td>
</tr>
<tr>
<td>+</td>
<td>#045h</td>
<td>69</td>
</tr>
<tr>
<td>-</td>
<td>#046h</td>
<td>70</td>
</tr>
<tr>
<td>*</td>
<td>#047h</td>
<td>71</td>
</tr>
<tr>
<td>/</td>
<td>#048h</td>
<td>72</td>
</tr>
<tr>
<td>^</td>
<td>#049h</td>
<td>73</td>
</tr>
<tr>
<td>&lt;</td>
<td>#0EBh</td>
<td>235</td>
</tr>
<tr>
<td>&gt;</td>
<td>#0EDh</td>
<td>237</td>
</tr>
<tr>
<td>«</td>
<td>#0ECb</td>
<td>236</td>
</tr>
<tr>
<td>»</td>
<td>#0EEh</td>
<td>238</td>
</tr>
<tr>
<td>=</td>
<td>#03Bh</td>
<td>59</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>#0E9h</td>
<td>233</td>
</tr>
<tr>
<td>#</td>
<td>#0E8h</td>
<td>234</td>
</tr>
<tr>
<td>#</td>
<td>#063h</td>
<td>99</td>
</tr>
<tr>
<td>+</td>
<td>#0FCh</td>
<td>252</td>
</tr>
<tr>
<td>+</td>
<td>#0FDh</td>
<td>253</td>
</tr>
<tr>
<td>:</td>
<td>#0F7h</td>
<td>247</td>
</tr>
<tr>
<td>+</td>
<td>#0F6h</td>
<td>248</td>
</tr>
<tr>
<td>=</td>
<td>#07Ch</td>
<td>124</td>
</tr>
<tr>
<td>=</td>
<td>#03Fh</td>
<td>63</td>
</tr>
<tr>
<td>+</td>
<td>#0FEh</td>
<td>254</td>
</tr>
<tr>
<td>-</td>
<td>#11Eh</td>
<td>296</td>
</tr>
<tr>
<td>=</td>
<td>#11Fh</td>
<td>287</td>
</tr>
<tr>
<td>-</td>
<td>#04Fh</td>
<td>79</td>
</tr>
<tr>
<td>=</td>
<td>#0FFh</td>
<td>255</td>
</tr>
<tr>
<td>=</td>
<td>#100h</td>
<td>256</td>
</tr>
<tr>
<td>=</td>
<td>#108h</td>
<td>267</td>
</tr>
<tr>
<td>=</td>
<td>#104h</td>
<td>268</td>
</tr>
<tr>
<td>=</td>
<td>#185h</td>
<td>261</td>
</tr>
<tr>
<td>=</td>
<td>#106h</td>
<td>262</td>
</tr>
</tbody>
</table>

Alphabetized for library #700h:

- CFI
  - CODE
  - CCRE
  - DIF
  - DO
  - ELSE
  - END
  - END
  - END
  - FOR
  - GROB
  - HALT
  - IF
  - IF
  - NEXT
  - PROMPT
  - REPEAT
  - START
  - STEP
  - THEN
  - THEN
  - THEN
  - UNTIL
  - WHILE
  - XLIB
  - +
  - +
  - -
  - +
  - <
  - >
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  - 

C. RPL Commands 349
The numerical table for library #002h:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000h</td>
<td>#157Bh ASR</td>
<td>051h</td>
<td>#1A5Ah FREEZE</td>
</tr>
<tr>
<td>001h</td>
<td>#159Bh RL</td>
<td>052h</td>
<td>#1A5Ch BEEP</td>
</tr>
<tr>
<td>002h</td>
<td>#15BBh RLB</td>
<td>053h</td>
<td>#1A54h +NUM</td>
</tr>
<tr>
<td>003h</td>
<td>#15D6h RR</td>
<td>054h</td>
<td>#1A64h LAST</td>
</tr>
<tr>
<td>004h</td>
<td>#15FBh RRB</td>
<td>055h</td>
<td>#1A67h LASTARG</td>
</tr>
<tr>
<td>005h</td>
<td>#161Bh SL</td>
<td>056h</td>
<td>#1A65h CLLCD</td>
</tr>
<tr>
<td>006h</td>
<td>#163Bh SLB</td>
<td>057h</td>
<td>#1A83h KEY</td>
</tr>
<tr>
<td>007h</td>
<td>#165Bh SRL</td>
<td>058h</td>
<td>#1A89h CONT</td>
</tr>
<tr>
<td>008h</td>
<td>#167Bh SRB</td>
<td>059h</td>
<td>#1A80h =</td>
</tr>
<tr>
<td>009h</td>
<td>#169Bh R+B</td>
<td>060h</td>
<td>#1A95h NEG</td>
</tr>
<tr>
<td>010h</td>
<td>#16BBh B+R</td>
<td>061h</td>
<td>#1A1Fh ABS</td>
</tr>
<tr>
<td>011h</td>
<td>#167Fh TIME</td>
<td>062h</td>
<td>#1A6Eh CONJ</td>
</tr>
<tr>
<td>012h</td>
<td>#1820h TICKS</td>
<td>063h</td>
<td>#1A18h x</td>
</tr>
<tr>
<td>013h</td>
<td>#1848h WSLOG</td>
<td>064h</td>
<td>#1A0Fh MATH</td>
</tr>
<tr>
<td>014h</td>
<td>#1863h ACKALL</td>
<td>065h</td>
<td>#1A81h MIN</td>
</tr>
<tr>
<td>015h</td>
<td>#1876h ACK</td>
<td>066h</td>
<td>#1A33h e</td>
</tr>
<tr>
<td>016h</td>
<td>#1889h +DATE</td>
<td>067h</td>
<td>#1A45h i</td>
</tr>
<tr>
<td>017h</td>
<td>#188Fh +TIME</td>
<td>068h</td>
<td>#1A46h +</td>
</tr>
<tr>
<td>018h</td>
<td>#180Dh CLRDJ</td>
<td>069h</td>
<td>#1A3Dh +</td>
</tr>
<tr>
<td>019h</td>
<td>#180Eh CTRL</td>
<td>070h</td>
<td>#1A09h -</td>
</tr>
<tr>
<td>020h</td>
<td>#180Fh STOALARM</td>
<td>071h</td>
<td>#1A0Eh *</td>
</tr>
<tr>
<td>021h</td>
<td>#181Ah FINDALARM</td>
<td>072h</td>
<td>#1A85h /</td>
</tr>
<tr>
<td>022h</td>
<td>#181Bh TIME</td>
<td>073h</td>
<td>#1B20h</td>
</tr>
<tr>
<td>023h</td>
<td>#181Ch DELALARM</td>
<td>074h</td>
<td>#1B19h XROOT</td>
</tr>
<tr>
<td>024h</td>
<td>#1932h TSTR</td>
<td>075h</td>
<td>#1B27h INV</td>
</tr>
<tr>
<td>025h</td>
<td>#1933h TSTR</td>
<td>076h</td>
<td>#1B28h ARG</td>
</tr>
<tr>
<td>026h</td>
<td>#1934h TSTR</td>
<td>077h</td>
<td>#1B29h SIGN</td>
</tr>
<tr>
<td>027h</td>
<td>#193Ah TSTR</td>
<td>078h</td>
<td>#1B3Ah SIGN</td>
</tr>
<tr>
<td>028h</td>
<td>#193Bh TSTR</td>
<td>079h</td>
<td>#1A34h</td>
</tr>
<tr>
<td>029h</td>
<td>#1A15h CROIR</td>
<td>080h</td>
<td>#1B42h SQ</td>
</tr>
<tr>
<td>030h</td>
<td>#1A16h CROIR</td>
<td>081h</td>
<td>#1B44h SIN</td>
</tr>
<tr>
<td>031h</td>
<td>#1A25h CROIR</td>
<td>082h</td>
<td>#1B5Ah SQ</td>
</tr>
<tr>
<td>032h</td>
<td>#1A26h CROIR</td>
<td>083h</td>
<td>#1B53Ah TAN</td>
</tr>
<tr>
<td>033h</td>
<td>#1A27h CROIR</td>
<td>084h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>034h</td>
<td>#1A28h CROIR</td>
<td>085h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>035h</td>
<td>#1A48h CROIR</td>
<td>086h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>036h</td>
<td>#1A58h CROIR</td>
<td>087h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>037h</td>
<td>#1A68h CROIR</td>
<td>088h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>038h</td>
<td>#1A78h CROIR</td>
<td>089h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>039h</td>
<td>#1A88h CROIR</td>
<td>090h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>040h</td>
<td>#1A98h CROIR</td>
<td>091h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>041h</td>
<td>#1A9Ah CROIR</td>
<td>092h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>042h</td>
<td>#1A9Bh CROIR</td>
<td>093h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>043h</td>
<td>#1A9Ah CROIR</td>
<td>094h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>044h</td>
<td>#1A9Ah CROIR</td>
<td>095h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>045h</td>
<td>#1A9Ah CROIR</td>
<td>096h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>046h</td>
<td>#1A9Ah CROIR</td>
<td>097h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>047h</td>
<td>#1A9Ah CROIR</td>
<td>098h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>048h</td>
<td>#1A9Ah CROIR</td>
<td>099h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>049h</td>
<td>#1A9Ah CROIR</td>
<td>100h</td>
<td>#1B5Ah TAN</td>
</tr>
<tr>
<td>050h</td>
<td>#1A9Ah CROIR</td>
<td>101h</td>
<td>#1B5Ah TAN</td>
</tr>
</tbody>
</table>

APPENDICES
<table>
<thead>
<tr>
<th>Command</th>
<th>RPL Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>#06h</td>
<td>Floor Plan</td>
</tr>
<tr>
<td>FLOOR</td>
<td>#07h</td>
<td></td>
</tr>
<tr>
<td>CEIL</td>
<td>#08h</td>
<td></td>
</tr>
<tr>
<td>TRNC</td>
<td>#09h</td>
<td></td>
</tr>
<tr>
<td>MOD</td>
<td>#0Eh</td>
<td>Modulus</td>
</tr>
<tr>
<td>DET</td>
<td>#0Fh</td>
<td>Detection</td>
</tr>
<tr>
<td>IMP</td>
<td>#10h</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>#13h</td>
<td></td>
</tr>
<tr>
<td>SUBP</td>
<td>#14h</td>
<td></td>
</tr>
<tr>
<td>POS</td>
<td>#16h</td>
<td>Position</td>
</tr>
<tr>
<td>STR</td>
<td>#17h</td>
<td></td>
</tr>
<tr>
<td>STR+</td>
<td>#18h</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>#19h</td>
<td></td>
</tr>
<tr>
<td>OBJ</td>
<td>#1Ah</td>
<td>Objective</td>
</tr>
<tr>
<td>OBJ+</td>
<td>#1Bh</td>
<td></td>
</tr>
<tr>
<td>ADDR</td>
<td>#1Ch</td>
<td>Address</td>
</tr>
<tr>
<td>ADDR+</td>
<td>#1Dh</td>
<td></td>
</tr>
<tr>
<td>ADDR-</td>
<td>#1Eh</td>
<td></td>
</tr>
<tr>
<td>ADDR-+</td>
<td>#1Fh</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>#22h</td>
<td>Fixed</td>
</tr>
<tr>
<td>ENG</td>
<td>#23h</td>
<td>English</td>
</tr>
<tr>
<td>STD</td>
<td>#24h</td>
<td>Standard</td>
</tr>
<tr>
<td>STD+</td>
<td>#25h</td>
<td></td>
</tr>
<tr>
<td>STD-</td>
<td>#26h</td>
<td></td>
</tr>
<tr>
<td>STD-+</td>
<td>#27h</td>
<td></td>
</tr>
</tbody>
</table>
APPENDICES
C. RPL Commands

Numerical table for library #700h:

| 0  | #000h | #22EC3h | IF    |
| 1  | #001h | #22EF0h | THEN  |
| 2  | #002h | #22FB5h | ELSE  |
| 3  | #003h | #22FD5h | END   |
| 4  | #004h | #22FFBh | +     |
| 5  | #005h | #23033h | WHILE |
| 6  | #006h | #23050h | REPEAT|
| 7  | #007h | #230C3h | DO    |
| 8  | #008h | #230Edh | UNTIL |
| 9  | #009h | #23183h | START |
| 10 | #00Ah | #231Afh | FOR   |
| 11 | #00Bh | #2324Ch | NEXT  |
| 12 | #00Ch | #23390h | STEP  |
| 13 | #00Dh | #2330Fh | IFERR |
| 14 | #00Eh | #23472h | HALT  |
| 15 | #00Fh | #2349Ch |      |
| 16 | #010h | #2341Ch |      |
| 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 | #23780h | CASE  |
| 26 | #01Ah | #23798h | THEN  |
| 27 | #01Bh | #23813h | C    |
| 28 | #01Ch | #23824h | PROMPT|

[Table continues with various library numbers and commands]
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#03FEFh</td>
<td>0</td>
</tr>
<tr>
<td>#03FF9h</td>
<td>1</td>
</tr>
<tr>
<td>#04003h</td>
<td>2</td>
</tr>
<tr>
<td>#0400Bh</td>
<td>3</td>
</tr>
<tr>
<td>#04017h</td>
<td>4</td>
</tr>
<tr>
<td>#04021h</td>
<td>5</td>
</tr>
<tr>
<td>#04029h</td>
<td>6</td>
</tr>
<tr>
<td>#04035h</td>
<td>7</td>
</tr>
<tr>
<td>#04045h</td>
<td>8</td>
</tr>
<tr>
<td>#04053h</td>
<td>9</td>
</tr>
<tr>
<td>#0405Dh</td>
<td>10</td>
</tr>
<tr>
<td>#04067h</td>
<td>11</td>
</tr>
<tr>
<td>#04071h</td>
<td>12</td>
</tr>
<tr>
<td>#0407Bh</td>
<td>13</td>
</tr>
<tr>
<td>#04085h</td>
<td>14</td>
</tr>
<tr>
<td>#04099h</td>
<td>15</td>
</tr>
<tr>
<td>#0409Dh</td>
<td>16</td>
</tr>
<tr>
<td>#040A7h</td>
<td>17</td>
</tr>
<tr>
<td>#040B1h</td>
<td>18</td>
</tr>
<tr>
<td>#040B9h</td>
<td>19</td>
</tr>
<tr>
<td>#040C3h</td>
<td>20</td>
</tr>
<tr>
<td>#040C9h</td>
<td>21</td>
</tr>
<tr>
<td>#040D5h</td>
<td>22</td>
</tr>
<tr>
<td>#040E5h</td>
<td>23</td>
</tr>
<tr>
<td>#040F0h</td>
<td>24</td>
</tr>
<tr>
<td>#040F9h</td>
<td>25</td>
</tr>
<tr>
<td>#040F3h</td>
<td>26</td>
</tr>
<tr>
<td>#040FDh</td>
<td>27</td>
</tr>
<tr>
<td>#04107h</td>
<td>28</td>
</tr>
<tr>
<td>#04111h</td>
<td>29</td>
</tr>
<tr>
<td>#0411Bh</td>
<td>30</td>
</tr>
<tr>
<td>#04125h</td>
<td>31</td>
</tr>
<tr>
<td>#0412Fh</td>
<td>32</td>
</tr>
<tr>
<td>#04139h</td>
<td>33</td>
</tr>
<tr>
<td>#04143h</td>
<td>34</td>
</tr>
<tr>
<td>#0414Dh</td>
<td>35</td>
</tr>
<tr>
<td>#04157h</td>
<td>36</td>
</tr>
<tr>
<td>#04161h</td>
<td>37</td>
</tr>
<tr>
<td>#0416Bh</td>
<td>38</td>
</tr>
<tr>
<td>#04175h</td>
<td>39</td>
</tr>
<tr>
<td>#0417Fh</td>
<td>40</td>
</tr>
<tr>
<td>#04183h</td>
<td>41</td>
</tr>
<tr>
<td>#04193h</td>
<td>42</td>
</tr>
<tr>
<td>#0419Dh</td>
<td>43</td>
</tr>
<tr>
<td>#041B2h</td>
<td>44</td>
</tr>
</tbody>
</table>

D. Objects in ROM
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1400h</td>
<td>1553</td>
</tr>
<tr>
<td>1402h</td>
<td>1554</td>
</tr>
<tr>
<td>1403h</td>
<td>1555</td>
</tr>
<tr>
<td>1406h</td>
<td>1556</td>
</tr>
<tr>
<td>1408h</td>
<td>1557</td>
</tr>
<tr>
<td>1416h</td>
<td>1558</td>
</tr>
<tr>
<td>1417h</td>
<td>1559</td>
</tr>
<tr>
<td>1418h</td>
<td>1560</td>
</tr>
<tr>
<td>1419h</td>
<td>1561</td>
</tr>
<tr>
<td>141ah</td>
<td>1562</td>
</tr>
<tr>
<td>141bh</td>
<td>1563</td>
</tr>
<tr>
<td>141Ch</td>
<td>1564</td>
</tr>
<tr>
<td>141Dh</td>
<td>1565</td>
</tr>
<tr>
<td>141eh</td>
<td>1566</td>
</tr>
<tr>
<td>1424h</td>
<td>1567</td>
</tr>
<tr>
<td>142eh</td>
<td>1568</td>
</tr>
<tr>
<td>1430h</td>
<td>1569</td>
</tr>
<tr>
<td>1432h</td>
<td>1570</td>
</tr>
<tr>
<td>1436h</td>
<td>1571</td>
</tr>
<tr>
<td>1439h</td>
<td>1572</td>
</tr>
<tr>
<td>143ah</td>
<td>1573</td>
</tr>
<tr>
<td>143eh</td>
<td>1574</td>
</tr>
<tr>
<td>1440h</td>
<td>1575</td>
</tr>
<tr>
<td>1446h</td>
<td>1576</td>
</tr>
<tr>
<td>144ah</td>
<td>1577</td>
</tr>
<tr>
<td>144eh</td>
<td>1578</td>
</tr>
<tr>
<td>144fh</td>
<td>1579</td>
</tr>
<tr>
<td>1458h</td>
<td>1580</td>
</tr>
<tr>
<td>1459h</td>
<td>1581</td>
</tr>
<tr>
<td>145eh</td>
<td>1582</td>
</tr>
<tr>
<td>1464h</td>
<td>1583</td>
</tr>
<tr>
<td>1465h</td>
<td>1584</td>
</tr>
<tr>
<td>1466h</td>
<td>1585</td>
</tr>
<tr>
<td>1467h</td>
<td>1586</td>
</tr>
<tr>
<td>1468h</td>
<td>1587</td>
</tr>
<tr>
<td>1469h</td>
<td>1588</td>
</tr>
<tr>
<td>146ah</td>
<td>1589</td>
</tr>
<tr>
<td>146eh</td>
<td>1590</td>
</tr>
<tr>
<td>1474h</td>
<td>1591</td>
</tr>
<tr>
<td>1475h</td>
<td>1592</td>
</tr>
<tr>
<td>1476h</td>
<td>1593</td>
</tr>
<tr>
<td>1477h</td>
<td>1594</td>
</tr>
<tr>
<td>1478h</td>
<td>1595</td>
</tr>
<tr>
<td>1479h</td>
<td>1596</td>
</tr>
<tr>
<td>147ah</td>
<td>1597</td>
</tr>
<tr>
<td>147eh</td>
<td>1598</td>
</tr>
<tr>
<td>1484h</td>
<td>1599</td>
</tr>
<tr>
<td>1485h</td>
<td>1600</td>
</tr>
<tr>
<td>1486h</td>
<td>1601</td>
</tr>
<tr>
<td>1487h</td>
<td>1602</td>
</tr>
<tr>
<td>1488h</td>
<td>1603</td>
</tr>
<tr>
<td>1489h</td>
<td>1604</td>
</tr>
<tr>
<td>148ah</td>
<td>1605</td>
</tr>
<tr>
<td>148eh</td>
<td>1606</td>
</tr>
<tr>
<td>1494h</td>
<td>1607</td>
</tr>
<tr>
<td>1495h</td>
<td>1608</td>
</tr>
<tr>
<td>1496h</td>
<td>1609</td>
</tr>
<tr>
<td>1497h</td>
<td>1610</td>
</tr>
<tr>
<td>1498h</td>
<td>1611</td>
</tr>
<tr>
<td>1499h</td>
<td>1612</td>
</tr>
<tr>
<td>149ah</td>
<td>1613</td>
</tr>
<tr>
<td>149eh</td>
<td>1614</td>
</tr>
<tr>
<td>14a4h</td>
<td>1615</td>
</tr>
<tr>
<td>14a5h</td>
<td>1616</td>
</tr>
<tr>
<td>14a6h</td>
<td>1617</td>
</tr>
<tr>
<td>14a7h</td>
<td>1618</td>
</tr>
<tr>
<td>14a8h</td>
<td>1619</td>
</tr>
<tr>
<td>14a9h</td>
<td>161a</td>
</tr>
<tr>
<td>14aa h</td>
<td>161b</td>
</tr>
<tr>
<td>14ab h</td>
<td>161c</td>
</tr>
<tr>
<td>14ac h</td>
<td>161d</td>
</tr>
<tr>
<td>14ad h</td>
<td>161e</td>
</tr>
<tr>
<td>14ae h</td>
<td>161f</td>
</tr>
<tr>
<td>14af h</td>
<td>1620</td>
</tr>
<tr>
<td>14b6h</td>
<td>1621</td>
</tr>
<tr>
<td>14b7h</td>
<td>1622</td>
</tr>
<tr>
<td>14b8h</td>
<td>1623</td>
</tr>
<tr>
<td>14b9h</td>
<td>1624</td>
</tr>
<tr>
<td>14ba h</td>
<td>1625</td>
</tr>
<tr>
<td>14bb h</td>
<td>1626</td>
</tr>
<tr>
<td>14bc h</td>
<td>1627</td>
</tr>
<tr>
<td>14bd h</td>
<td>1628</td>
</tr>
<tr>
<td>14be h</td>
<td>1629</td>
</tr>
<tr>
<td>14bf h</td>
<td>162a</td>
</tr>
<tr>
<td>14c0h</td>
<td>162b</td>
</tr>
<tr>
<td>14c1h</td>
<td>162c</td>
</tr>
<tr>
<td>14c2h</td>
<td>162d</td>
</tr>
<tr>
<td>14c3h</td>
<td>162e</td>
</tr>
<tr>
<td>14c4h</td>
<td>162f</td>
</tr>
<tr>
<td>14c5h</td>
<td>1630</td>
</tr>
<tr>
<td>14c6h</td>
<td>1631</td>
</tr>
<tr>
<td>14c7h</td>
<td>1632</td>
</tr>
<tr>
<td>14c8h</td>
<td>1633</td>
</tr>
<tr>
<td>14c9h</td>
<td>1634</td>
</tr>
<tr>
<td>14ca h</td>
<td>1635</td>
</tr>
<tr>
<td>14cb h</td>
<td>1636</td>
</tr>
<tr>
<td>14cc h</td>
<td>1637</td>
</tr>
<tr>
<td>14cd h</td>
<td>1638</td>
</tr>
<tr>
<td>14ce h</td>
<td>1639</td>
</tr>
<tr>
<td>14cf h</td>
<td>163a</td>
</tr>
<tr>
<td>14d0h</td>
<td>163b</td>
</tr>
<tr>
<td>14d1h</td>
<td>163c</td>
</tr>
<tr>
<td>14d2h</td>
<td>163d</td>
</tr>
<tr>
<td>14d3h</td>
<td>163e</td>
</tr>
<tr>
<td>14d4h</td>
<td>163f</td>
</tr>
<tr>
<td>14d5h</td>
<td>1640</td>
</tr>
<tr>
<td>14d6h</td>
<td>1641</td>
</tr>
<tr>
<td>14d7h</td>
<td>1642</td>
</tr>
<tr>
<td>14d8h</td>
<td>1643</td>
</tr>
<tr>
<td>14d9h</td>
<td>1644</td>
</tr>
<tr>
<td>14da h</td>
<td>1645</td>
</tr>
<tr>
<td>14db h</td>
<td>1646</td>
</tr>
<tr>
<td>14dc h</td>
<td>1647</td>
</tr>
<tr>
<td>14dd h</td>
<td>1648</td>
</tr>
<tr>
<td>14de h</td>
<td>1649</td>
</tr>
<tr>
<td>14df h</td>
<td>164a</td>
</tr>
<tr>
<td>14e0h</td>
<td>164b</td>
</tr>
<tr>
<td>14e1h</td>
<td>164c</td>
</tr>
<tr>
<td>14e2h</td>
<td>164d</td>
</tr>
<tr>
<td>14e3h</td>
<td>164e</td>
</tr>
<tr>
<td>14e4h</td>
<td>164f</td>
</tr>
<tr>
<td>14e5h</td>
<td>1650</td>
</tr>
<tr>
<td>14e6h</td>
<td>1651</td>
</tr>
<tr>
<td>14e7h</td>
<td>1652</td>
</tr>
<tr>
<td>14e8h</td>
<td>1653</td>
</tr>
<tr>
<td>14e9h</td>
<td>1654</td>
</tr>
<tr>
<td>14ea h</td>
<td>1655</td>
</tr>
<tr>
<td>14eb h</td>
<td>1656</td>
</tr>
<tr>
<td>14ec h</td>
<td>1657</td>
</tr>
<tr>
<td>14ed h</td>
<td>1658</td>
</tr>
<tr>
<td>14ee h</td>
<td>1659</td>
</tr>
<tr>
<td>14ef h</td>
<td>165a</td>
</tr>
<tr>
<td>14f0h</td>
<td>165b</td>
</tr>
<tr>
<td>14f1h</td>
<td>165c</td>
</tr>
<tr>
<td>14f2h</td>
<td>165d</td>
</tr>
<tr>
<td>14f3h</td>
<td>165e</td>
</tr>
<tr>
<td>14f4h</td>
<td>165f</td>
</tr>
<tr>
<td>14f5h</td>
<td>1660</td>
</tr>
<tr>
<td>14f6h</td>
<td>1661</td>
</tr>
<tr>
<td>14f7h</td>
<td>1662</td>
</tr>
<tr>
<td>14f8h</td>
<td>1663</td>
</tr>
<tr>
<td>14f9h</td>
<td>1664</td>
</tr>
<tr>
<td>14fa h</td>
<td>1665</td>
</tr>
<tr>
<td>14fb h</td>
<td>1666</td>
</tr>
<tr>
<td>14fc h</td>
<td>1667</td>
</tr>
<tr>
<td>14fd h</td>
<td>1668</td>
</tr>
<tr>
<td>14fe h</td>
<td>1669</td>
</tr>
<tr>
<td>14ff h</td>
<td>166a</td>
</tr>
</tbody>
</table>

D. Objects in ROM 357
Real Numbers

9.9999999999E499
-4.77451811461E441
-268
-9
-8
-7
-6
-5
-4
-3
-2
-1
-0.5
-1E-499
0
1E-499
1E-12
3.4906585399E-2
0.1
4.34294481904E-1
0.5
0.15
1
1.8
2
2.5
7.1828182946
3
3.14159265359
4
5
6
6.28318530718
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
40

Long Real Numbers

-1E-10000
-495.92811971593
-76.5594018140200
-1.21142857142857
0
1E-10000
1.74532925199433E-2
7.95774715494477E-2
0.1
0.4
0.5
5.5555555555556E-1
0.7
9.18938533204673E-1
1
2
2.30258509299405
3
3.14159265358979
4
5
6.28318530717959
7
9.33584965666377
10
12
36.3479606873615
32
60
100
273.15
459.67
1E10000
### Complex Numbers

<table>
<thead>
<tr>
<th>Code</th>
<th>Real</th>
<th>Imaginary</th>
</tr>
</thead>
<tbody>
<tr>
<td>#5182h</td>
<td>(0, 0)</td>
<td></td>
</tr>
<tr>
<td>#524Fh</td>
<td>(0, 0)</td>
<td></td>
</tr>
<tr>
<td>#5196h</td>
<td>(1, 0)</td>
<td></td>
</tr>
<tr>
<td>#524Fh</td>
<td>(1, 0)</td>
<td></td>
</tr>
<tr>
<td>#526Fh</td>
<td>(0, 1)</td>
<td></td>
</tr>
<tr>
<td>#526Fh</td>
<td>(0, 1)</td>
<td></td>
</tr>
<tr>
<td>#526Fh</td>
<td>(0, -1)</td>
<td></td>
</tr>
</tbody>
</table>

### Long Complex Numbers

<table>
<thead>
<tr>
<th>Code</th>
<th>Real</th>
<th>Imaginary</th>
</tr>
</thead>
<tbody>
<tr>
<td>#5193h</td>
<td>(0, 0)</td>
<td></td>
</tr>
</tbody>
</table>

### Characters

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>#5406h</td>
<td>'C'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'D'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'E'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'F'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'G'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'H'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'I'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'J'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'K'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'L'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'M'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'N'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'O'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'P'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'Q'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'R'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'S'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'T'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'U'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'V'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'W'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'X'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'Y'</td>
</tr>
<tr>
<td>#5406h</td>
<td>'Z'</td>
</tr>
</tbody>
</table>

---

D. Objects in ROM 359
Arrays

#72800h [ "Insufficient Memory" "Directory Recursion" "Undefined Local Name" "Undefined XLIB Name" "Memory Clear" "Power Lost" "Warning1" "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 Cad." "Port Closed" "Connecting" "Retry #" "Awaiting Server Cad." "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:" "Exeption:" "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 LCN(Neg)" "Invalid Σ Data LCN(0)" "Invalid ED" "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 1 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 PTYP£" "Name the stat data," "press ENTER"
"Enter value (zoom out if >1), press ENTER" "Copied to stack"
"X axis zoom w/AUTO." " Y axis zoom.
"X and Y axis zoom." "IR/wire: " "ASCII/binary: " "baud:"
"parity: " "checksum type: " "translate code:"
"Enter matrix, then NEW"

#736Fh [ "Invalid Dimension" "Invalid Array Element" "Deleting Row"
"Deleting Column" "Inserting Row" "Inserting Column" ]

#7AA50h [ <3F0Bh> <3F0Fh> <4000h> <6540h> <6559h> <7A939h> ]
#7AA94h [ <4003h> <4006h> <4009h> <6540h> <6559h> <7A968h> ]
#7AACBh [ <4007h> <400Ah> <400Dh> <6540h> <6559h> <7A96Fh> ]
#7AB02h [ <400Ch> <400Dh> <4009h> <6540h> <6559h> <7A976h> ]
#7AB39h [ <4010h> <4013h> <4019h> <6540h> <6559h> <7A976h> ]
#7AB70h [ <4015h> <4018h> <4019h> <6540h> <6559h> <7A970h> ]
#7ABA7h [ <3A05h> <3AE9h> <1EE53h> <6540h> <6559h> <7A984h> ]
#7ABD8h [ <3A09h> <3A09h> <3A0Dh> <6540h> <6559h> <7A992h> ]
#7AC15h [ <3A0Bh> <3A0Ah> <3A0Ah> <6540h> <6559h> <7A990h> ]
#7AC4Ch [ <3A0Fh> <3A0Ch> <3A0Hh> <6540h> <6559h> <7A9AFh> ]
#7ACB3h [ <3A0Eh> <3A0Fh> <3A06h> <6540h> <6559h> <7A997h> ]
#7AC8Ah [ <3A0Fh> <3A0Ah> <3A0Ah> <6540h> <6559h> <7A997h> ]
#7ACF1h [ <3A0Ah> <3A0Bh> <3A0Ah> <6540h> <6559h> <7A998h> ]
#7AD29h [ <3A0Ch> <3A0Ch> <3A0Ch> <6540h> <6559h> <7A997h> ]
#7ADF5h [ <1A08h> <1F9Ch> <1A5E4h> <6552h> <6555h> <7A90Ah> ]
#7AD96h [ <3A09h> <1E29h> <3A0Ah> <6553h> <655Eh> <7A90Bh> ]
#7ADDDh [ <3A04h> <3A0Ch> <3A04h> <6553h> <655Fh> <7A90Bh> ]
#7AE04h [ <3A0Fh> <1F8Ah> <3A0Ch> <6554h> <655Fh> <7A90Fh> ]
#7AE3Bh [ <1B4FCh> <1B6Ah> <1F7Fh> <6554h> <6560h> <7A9E6h> ]
#7AE72h [ <1B50h> <1B72h> <1F10h> <6555h> <6560h> <7A9EDh> ]
#7AE99h [ <1B55h> <1B79h> <1F2Ch> <6555h> <6560h> <7A93Bh> ]
#7AE90h [ <1B37h> <1B42h> <1B15h> <6556h> <6561h> <7A961h> ]
#7AF17h [ <1B02h> <1BA0h> <1B96h> <6557h> <6561h> <7A9FBh> ]
#7AF4eh [ <1B27h> <1B90h> <1B94h> <6556h> <6562h> <7A902h> ]

D. Objects in ROM 361
D. Objects in ROM

Strings

\#050h "\n\#052h "\n\#055h "\n\#06Eh "\n\#075h "\n\#077h "\n\#076h "\n\#079h "\n\#07Ah "\n\#07Dh "\n\#087h "\n\#089h "\n\#08Ah "\n\#08Ch "\n\#08Dh "\n\#089h "\n\#090h "\n\#096h "\n\#098h "\n\#09Ah "\n\#09Ch "\n\#09Dh "\n\#099h "\n\#0A6h "\n\#0A8h "\n\#0ACh "\n\#0ADh "\n\#0AFh "\n\#0BFh "\n\#0C0h "\n\#0C2h "\n\#0C4h "\n\#0C6h "\n\#0C8h "\n\#0Ca0h "\n\#0CCh "\n\#0CDh "\n\#0CFh "\n\#0DFh "\n\#0E0h "\n\#0E2h "\n\#0E4h "\n\#0E6h "\n\#0E8h "\n\#0ECh "\n\#0EDh "\n\#0EEh "\n\#0F0h "\n\#0F2h "\n\#0F4h "\n\#0F6h "\n\#0F8h "\n\#0FAh "\n\#0FC0h "\n\#0FCh "\n\#0FFh "\n
"TO"
"DIR"
";
"ELSE"
"END"
"UNTIL"
"REPEAT"
"NEXT"
"STEP"
"THEN"
"*"
"bodh"
"C5F3h"
"SORT"
"SQ"
"INV"
"Invalid Expression"
"<&HP>
"31026h"
"32341h"
"34185h"
"34155h"
"3412Fh"
"39710h"
"39709h"
"USER"
"39829h"
"ALG"
"PRG"
"3998Ah"

1

"39F80h"

39F28h

39FF2h

39920h

392C0h

392E1h

39300h

39323h

39346h

39558h

3957Ah

39599h

3958Ah

395D0h

395FC0h

3965Eh

397Edh

3980Dh

3982Ch

3984Dh

39A12h

39A33h

39B6Eh

39B85h
D. Objects in ROM
### Binary Integers

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#65769h</td>
<td>&quot;EXIT&quot;</td>
</tr>
<tr>
<td>1</td>
<td>#65770h</td>
<td>&quot;Undefined&quot;</td>
</tr>
<tr>
<td>2</td>
<td>#65771h</td>
<td>&quot;BAD&quot;</td>
</tr>
<tr>
<td>3</td>
<td>#65772h</td>
<td>&quot;GRAD&quot;</td>
</tr>
<tr>
<td>4</td>
<td>#65773h</td>
<td>&quot;d&quot;</td>
</tr>
<tr>
<td>5</td>
<td>#65774h</td>
<td>&quot;RATIO&quot;</td>
</tr>
<tr>
<td>6</td>
<td>#65775h</td>
<td>&quot;RULES&quot;</td>
</tr>
<tr>
<td>7</td>
<td>#65777h</td>
<td>&quot;EDIT&quot;</td>
</tr>
<tr>
<td>8</td>
<td>#65778h</td>
<td>&quot;EXPR&quot;</td>
</tr>
<tr>
<td>9</td>
<td>#65779h</td>
<td>&quot;SUB&quot;</td>
</tr>
<tr>
<td>10</td>
<td>#65780h</td>
<td>&quot;REPL&quot;</td>
</tr>
<tr>
<td>11</td>
<td>#65781h</td>
<td>&quot;NOT&quot;</td>
</tr>
<tr>
<td>12</td>
<td>#10E34h</td>
<td>00000000000000000h</td>
</tr>
<tr>
<td>13</td>
<td>#10E35h</td>
<td>00000000000000001h</td>
</tr>
<tr>
<td>14</td>
<td>#16215h</td>
<td>000000000000000000h</td>
</tr>
<tr>
<td>15</td>
<td>#16216h</td>
<td>000000000000000001h</td>
</tr>
<tr>
<td>16</td>
<td>#16217h</td>
<td>000000000000000010h</td>
</tr>
<tr>
<td>17</td>
<td>#16218h</td>
<td>000000000000000011h</td>
</tr>
<tr>
<td>18</td>
<td>#16219h</td>
<td>000000000000000100h</td>
</tr>
<tr>
<td>19</td>
<td>#1621Ah</td>
<td>000000000000000101h</td>
</tr>
<tr>
<td>20</td>
<td>#1621Bh</td>
<td>000000000000000110h</td>
</tr>
<tr>
<td>21</td>
<td>#1621Ch</td>
<td>000000000000000111h</td>
</tr>
<tr>
<td>22</td>
<td>#16220h</td>
<td>000000000000001000h</td>
</tr>
<tr>
<td>23</td>
<td>#16221h</td>
<td>000000000000001001h</td>
</tr>
<tr>
<td>24</td>
<td>#16222h</td>
<td>000000000000001010h</td>
</tr>
<tr>
<td>25</td>
<td>#16223h</td>
<td>000000000000001011h</td>
</tr>
<tr>
<td>26</td>
<td>#16224h</td>
<td>000000000000001100h</td>
</tr>
<tr>
<td>27</td>
<td>#16225h</td>
<td>000000000000001101h</td>
</tr>
<tr>
<td>28</td>
<td>#16226h</td>
<td>000000000000001110h</td>
</tr>
<tr>
<td>29</td>
<td>#16227h</td>
<td>000000000000001111h</td>
</tr>
</tbody>
</table>

**D. Objects in ROM**
Lists

055E9h ( )
065A0h ( )
0E475h ( M N )
0FA53h ( '1_kg' '1_m' '1_A'
'1_s' '1_K' '1_cd' '1_mol'
'1_?')
10105h ( )
1085Eh ( )
10866h ( )
10872h ( )
1087Ch ( )
10886h ( )
14396h ( 'halt' )
155EFh ( 'nohalt' )
19991h ( 0 " 0 )
19F8Ah ( 4954521600 787788800
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" )
22340h ( 1208 2108 4000 9600 )
223C9h ( 1 2 3 )
22400h ( 0 1 2 3 4 )
22441h ( 0 1 2 3 )
23754h ( 'noname' 'stop' )
23B79h ( 'joinprogress' )
23903h ( st of s tok )
23999h ( )
24A28h ( 1 J )
25699h ( <zh> <zh> <zh> <zh> <zh> )
25BEEh ( )
25906h ( '1 '2 '3 ' )
272CBh ( 'tit str of s tok rbv'
'iffig tmpop tpmpdat'
'ploc bv unbound' )
2807Rh ( )
28103h ( 1 * + )
28884h ( )
2C756h ( 1 2 0 0 LINFIT )
2D40Bh ( )
2D5F5h ( 'PACKET 'RETRY 'MaxR'
'KP 'PKNO' )
2D839h ( 'KP 'PKNO' )
2D868h ( 'LNAME 'RETRY 'KMODE'
'KRM' )
2D056h ( 'ERRMSG' )
2DF01h ( 'LNAME 'KMODE 'KRM' )

#2E6CDh ( 'KP 'PKNO' )
#2E855h ( 'KP 'PKNO' )
#2E9B0h ( 'LNAME 'KMODE 'KRM' )
#2E99Eh ( 9608 0 0 0 3 1 )
#2E9F0h ( )
#2EE03h ( 'LNAME 'OBJ 'PACKET'
'KRM' )
#2F1FDh ( 'KLIST 'OPOS 'KML' )
#2F6F8h ( 'RETRY' )
#31C32h ( 'IWrap' )
#31C99h ( 1 )
#31F4Ah ( 1.8 " 80 " ' )
#32FF9h ( 'nohalt' )
#3304Eh ( <zh> )
#3306Ch ( )
#3402Bh ( )
#36796h ( <zh> )
#368CCh ( 3 )
#36F1h ( # a # )
#36F5Ah ( # )
#3605Ah ( )
#36082h ( )
#38294h ( 'SavedUI' )
#38290h ( "PARTS" ( ABS SIGN CONJ
ARG RE IM MIN MAX MOD %
+CH +T MANT XPON IP FP
FLOOR CEIL RND TRNC MAKR
MINR ) )
#3820Ch ( "HYP" ( SINH ASINH COSH
ACOSH TANH ATANH EXPM
LN1 ) )
#382FBh ( "MATRIX" ( CON IDN TRN
ROM DET RSD ABS RNRM CNRM
) )
#3836Ch ( ABS SIGN CONJ ARG RE IM
MIN MAX MOD % +CH +T MANT
XPON IP FP FLOOR CEIL RND
TRNC MAKR MINR )
#38A2Bh ( SINH ASINH COSH
ACOSH TANH ATANH EXPM
LN1 )
#38452h ( CON IDN TRN ROM DET RSD
ABS RNRM CNRM )
#38556h ( "STK" ( OVER ROT ROLL
ROLD PICK DEPTH DUP DUP2
DUPN DROP2 ( "DRPN" DROPN
) )
#38575h ( "OBJ" ( OBJ+ EQ+ ARRAY
+LIST +STR +TAG R+C C+R
DTAG +UNIT TYPE VTYPE SIZE
POS REPL SUB NUM CHR
PUT GET PUTI GETI ) )
Units

- 1_kg'
- 1_m'
- 1_A'
- 1_s'
- 1_K'
- 1_cd'
- 1_mol'
- 1_?'
Graphics Objects

#130B4h GROB 6 10 F1F1F1F1F1F1F1F100

#39B2Dh GROB 131 2 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

#3A337h GROB 21 8 FFFFF0000000000000000000000000000

#3A399h GROB 21 8 FFFFF0000000000000000000000000000

#3A3FBh GROB 21 8 1CFFFF0000000000000000000000000000

#3A45Dh GROB 21 8 FFFFF1000011000E1100011000110001100

#5853Ch GROB 5 5 4040F14040

#5855Ah GROB 5 5 11A040A011

#585B2h GROB 8 8 .

#66EA5h GROB 6 10 F1111111111111111F100

#66ECDh GROB 6 8 F111111111111111F100

#66EF1h GROB 4 6 F890909090F0

#66F11h GROB 6 8 0000000000000000

#66F35h GROB 6 10 0000000000000000

#66F5Dh GROB 7 5 F77D605F7F7

#66F7Dh GROB 5 4 F1B151F1

D. Objects in ROM
Global Names

#00B81h 'Alarms'
#00B29h 'Alarms'
#1576Ch 'EQ'
#15781h 's
#19A72h 'ALRMAT'
#1981Fh 'ALRMAT'
#1908Ch 'ALRMAT'
#21184h 'CST'
#225A4h 'DAT'
#225F0h 'DAT'
#2C796h 'IPAR'
#2C905h 'IPAR'
#2E6F9h 'IPAR'
#31FB2h 'PRPAR'
#31FB8h 'PRPAR'
#34088h 'symb'
#3FAF6h 'SKEY'
#40939h 'ENTER'
#409DFh 'ENTER'
#4A913h 'UserKeys'
#4A959h 'UserKeys.CRC'
#4B077h 's
#5393Ch 'ALG'
#5398Ch 'a'
#53CC6h 'x'
#54F39h 'ALRMAT'
#5A145h 'x'
#5A196h 'x'
#5A10Eh 'x'
#5A220h 'x'
#5A25Eh 'x'
#5A81Ch 'x'
#5A859h 'x'
#5B0CEh 'x'
#5B0E6h 'x'
#51286h 'PPAR'
#51436h 's
#56659h 'ERR'
#5729Fh 'nB'
#57950h 's0'
#59384h 's1'

Local Names

#0E47Ah 'M'
#0E483h 'N'
#0E4A6h 'M'
#0E4AEh 'N'
#0E4CHh 'M'
#14398h 'halt'
#14483h 'nohalt'
#1F96Fh 'num'
#1F97Eh 'fcn'
#2372Eh 'stop'
#2373Fh 'noname'

#2387Eh 'joinprogress'
#23908h 'st'
#23913h 'ofs'
#23928h 'tok'
#23949h 'st'
#23956h 'ofs'
#23963h 'tok'
#24240h 'j'
#24256h 'j'
#2425Eh 'j'
#2488Ah 'j'
#24B1Dh 'j'
#24B39h 'j'
#24B9Eh 'j'
#24BD3h '1'
#25088h '2'
#25A16h '3'
#25A21h '3'
#25A96h '1'
#25A46h '2'
#25A51h '3'
#272CDh 'ttl'
#2720Ch 'str'
#272E8h 'ofs'
#272FAh 'tok'
#27369h 'rbv'
#2731Bh 'idflg'
#27320h 'tmpop'
#2734Bh 'tmpdat'
#27357h 'loc'
#27368h 'bu'
#273D5h 'unbound'
#2B9A8h 'PKNO'
#2B918h 'PACKET'
#2B956h 'RETRY'
#2B969h 'ERRMSG'
#2B9EEh 'KP'
#210FBh 'LNAME'
#2049Eh 'OBJ'
#2041Dh 'OPDS'
#2042Ch 'EXCHP'
#2045Ah 'KLIST'
#2046Dh 'KMODE'
#2049Eh 'KPTRN'
#20493h 'KRM'
#204ARh 'MaxR'
#2F211h 'KML'
#2F46Ch 'KEOF'
#31C37h 'lwrap'
#34D96h 'savedUI'
#368F6h '#a'
#36C81h '#b'
#36C26h '#b'
#36C3Fh '#a'
#36CE6h '#b'
#36D19h '#b'
#39A3Eh 'SavedUI'
#3FAE9h 'SKEY'

APPENDICES
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>001h</td>
<td>&quot;Insufficient Memory&quot;</td>
</tr>
<tr>
<td>002h</td>
<td>&quot;Directory Recursion&quot;</td>
</tr>
<tr>
<td>003h</td>
<td>&quot;Undefined Local Name&quot;</td>
</tr>
<tr>
<td>004h</td>
<td>&quot;Undefined XLIB Name&quot;</td>
</tr>
<tr>
<td>005h</td>
<td>&quot;Memory Clear&quot;</td>
</tr>
<tr>
<td>006h</td>
<td>&quot;Power Lost&quot;</td>
</tr>
<tr>
<td>007h</td>
<td>&quot;Warning!&quot;</td>
</tr>
<tr>
<td>008h</td>
<td>&quot;Invalid Card Data&quot;</td>
</tr>
<tr>
<td>009h</td>
<td>&quot;Object In Use&quot;</td>
</tr>
<tr>
<td>00Ah</td>
<td>&quot;Port Not Available&quot;</td>
</tr>
<tr>
<td>00Bh</td>
<td>&quot;No Room in Port&quot;</td>
</tr>
<tr>
<td>00Ch</td>
<td>&quot;Object Not in Port&quot;</td>
</tr>
<tr>
<td>00Dh</td>
<td>&quot;Recovering Memory&quot;</td>
</tr>
<tr>
<td>00Eh</td>
<td>&quot;Try To Recover Memory?&quot;</td>
</tr>
<tr>
<td>00Fh</td>
<td>&quot;Replace RAM, Press ON&quot;</td>
</tr>
<tr>
<td>010h</td>
<td>&quot;No Mem To Config All&quot;</td>
</tr>
<tr>
<td>011h</td>
<td>&quot;No Room to Save Stack&quot;</td>
</tr>
<tr>
<td>012h</td>
<td>&quot;Can't Edit Null Char.&quot;</td>
</tr>
<tr>
<td>013h</td>
<td>&quot;Invalid User Function&quot;</td>
</tr>
<tr>
<td>014h</td>
<td>&quot;No Current Equation&quot;</td>
</tr>
<tr>
<td>016h</td>
<td>&quot;Invalid Syntax&quot;</td>
</tr>
<tr>
<td>017h</td>
<td>&quot;Real Number&quot;</td>
</tr>
<tr>
<td>018h</td>
<td>&quot;Complex Number&quot;</td>
</tr>
<tr>
<td>019h</td>
<td>&quot;String&quot;</td>
</tr>
<tr>
<td>01Ah</td>
<td>&quot;Real Array&quot;</td>
</tr>
<tr>
<td>01Bh</td>
<td>&quot;Complex Array&quot;</td>
</tr>
<tr>
<td>01Ch</td>
<td>&quot;List&quot;</td>
</tr>
<tr>
<td>01Dh</td>
<td>&quot;Global Name&quot;</td>
</tr>
<tr>
<td>01Eh</td>
<td>&quot;Local Name&quot;</td>
</tr>
<tr>
<td>01Fh</td>
<td>&quot;Program&quot;</td>
</tr>
<tr>
<td>020h</td>
<td>&quot;Algebraic&quot;</td>
</tr>
<tr>
<td>021h</td>
<td>&quot;Binary Integer&quot;</td>
</tr>
<tr>
<td>024h</td>
<td>&quot;LAST STACK Disabled&quot;</td>
</tr>
<tr>
<td>025h</td>
<td>&quot;LAST CMD Disabled&quot;</td>
</tr>
<tr>
<td>026h</td>
<td>&quot;HALT Not Allowed&quot;</td>
</tr>
<tr>
<td>027h</td>
<td>&quot;Array&quot;</td>
</tr>
<tr>
<td>028h</td>
<td>&quot;Wrong Argument Count&quot;</td>
</tr>
<tr>
<td>029h</td>
<td>&quot;Circular Reference&quot;</td>
</tr>
<tr>
<td>02Ah</td>
<td>&quot;Directory Not Allowed&quot;</td>
</tr>
<tr>
<td>02Bh</td>
<td>&quot;Non-Empty Directory&quot;</td>
</tr>
<tr>
<td>02Ch</td>
<td>&quot;Invalid Definition&quot;</td>
</tr>
<tr>
<td>02Dh</td>
<td>&quot;Missing Library&quot;</td>
</tr>
<tr>
<td>02Eh</td>
<td>&quot;Invalid PPAR&quot;</td>
</tr>
<tr>
<td>02Fh</td>
<td>&quot;Non-Real Result&quot;</td>
</tr>
<tr>
<td>030h</td>
<td>&quot;Unable to Isolate&quot;</td>
</tr>
<tr>
<td>031h</td>
<td>&quot;No Room to Show Stack&quot;</td>
</tr>
<tr>
<td>032h</td>
<td>&quot;Warning!&quot;</td>
</tr>
<tr>
<td>033h</td>
<td>&quot;Error!&quot;</td>
</tr>
<tr>
<td>034h</td>
<td>&quot;Purge?&quot;</td>
</tr>
<tr>
<td>035h</td>
<td>&quot;Out of Memory&quot;</td>
</tr>
<tr>
<td>036h</td>
<td>&quot;Stack&quot;</td>
</tr>
<tr>
<td>037h</td>
<td>&quot;Last Stack&quot;</td>
</tr>
<tr>
<td>038h</td>
<td>&quot;Last Commands&quot;</td>
</tr>
<tr>
<td>039h</td>
<td>&quot;Key Assignments&quot;</td>
</tr>
<tr>
<td>03Ah</td>
<td>&quot;Alarms&quot;</td>
</tr>
<tr>
<td>03Bh</td>
<td>&quot;Last Arguments&quot;</td>
</tr>
<tr>
<td>03Ch</td>
<td>&quot;Name Conflict&quot;</td>
</tr>
<tr>
<td>03Dh</td>
<td>&quot;Command Line&quot;</td>
</tr>
</tbody>
</table>

E. Error Messages 375
APPENDICES
E. Error Messages

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"

3329 \# D01h "Invalid Date"
3330 \# D02h "Invalid Time"
3331 \# D03h "Invalid Repeat"
3332 \# D04h "Nonexistent Alarm"

458752 \# 70000h Last user message (message DOERR)

3873 \# C01h "Bad Packet Block Check"
3874 \# C02h "Timeout"
3875 \# C03h "Receive Error"
3876 \# C04h "Receive Buffer Overrun"
3877 \# C05h "Parity Error"
3878 \# C06h "Transfer Failed"
3879 \# C07h "Protocol Error"
3880 \# C08h "Invalid Server Cmd."
3881 \# C09h "Port Closed"
3882 \# C0Ah "Connecting"
3883 \# C0Bh "Retry #"
3884 \# C0Ch "Awaiting Server Cmd."
3885 \# C0Dh "Sending"
3886 \# C0Eh "Receiving"
3887 \# C0Fh "Object Discarded"
3888 \# C10h "Packet #"
3889 \# C11h "Processing Command"
3890 \# C12h "Invalid IOPAR"
3891 \# C13h "Invalid PRTPAR"
3892 \# C14h "Low Battery"
3893 \# C15h "Empty Stack"
3894 \# C16h "Row"
3895 \# C17h "Invalid Name"
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>
<thead>
<tr>
<th>Field</th>
<th>a</th>
<th>f</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>WP</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>XS</td>
<td>2</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>X</td>
<td>3</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>S</td>
<td>4</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>M</td>
<td>5</td>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>6</td>
<td>E</td>
</tr>
<tr>
<td>W</td>
<td>7</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
F. Machine Language Instructions

381
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, a, x, 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 (Binary Coded Decimal)** 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 buffer, and the data going out or coming in the RS232c port goes through a buffer.

**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 (Liquid Crystal Display)**  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 (Random Access Memory)**  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 (Read Only Memory)**  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 GOSBYL.

- **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 (# 039BEn)** 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_R® (@ # 06537n)** 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_R** (#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.

• **DIVS** (#06A8EN) 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 than 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 battery 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.

• **RDISPOFF** (#01BD3h) turns off the display and the annunciators.

• **DISPON** (#01B8Fh) turns on the display.

• **RDISPON** (#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 \[((\text{size in pixels}) + 7 ) / 4\].

• **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.

• **BUSYN0** (\#42359h) turns off the BUSY annunciator.
I. Index
2's complement 83, 110
?ADR 228
mSOLVER 295
¥ 286

@ of alarms 196
@ of backup area 187, 194
@ of command line 191
@ of current GROB 186
@ of last error message 195
@ of menu GROB 186
@ of next object to execute 197
@ of PICT GROB 186
@ of stack GROB 186
@ of temporary environment 186, 193
@ of the current directory 194
@ of the End of RAM 195
@ of the home directory 194
@ of the undo stack and local vars 191
@ of user-keys 196
@i 66

A 77, 78
A->STR 260
A->V 289
Absolute 84, 111, 112
ADD 283
Addition 83, 98
Address 66, 77, 243, 383

Address of Last Error Message 195
Address of an object in level n 200
Address of Hash Table 136
Address of Message Table 136
ADISPOFF 390
ADISPON 390
Alarms 196
Alert 164
Algebraic object 123, 139
ALLBYTES 216
ALLINTR 390
Alpha 164
ANAG 307
Annunciators 164, 201, 202, 383
Arithmetic operations 83, 97
Arrays 360
ASCII code 129
ASLW5 389
ASN 50
ASRW5 389
Assemble 383
Assembler 69, 383
Assembling 70
Assembly 70
Attn Flag 201
Auto-test fail time 176, 178
Auto-test start time 176, 178
AUTOST 319

B 77, 78, 175, 186, 210
B->SB 258
BACKUP 54, 145, 175, 194
Backup Area 194
Backup End 194
Backup object 123, 148
Backups 194
Bank-switching 159, 383
BANNER 324
Base 341
Base address of built-in RAM 164, 170
Batteries 164, 177
Battery Test 164
BCD 71, 125, 383
BEEP 73, 388
Beginning @ of free mem. 186
Beginning @ of temporary objects 186
BFAC 283
BFREE 261
Binary 342
Binary coded decimal 71, 125, 383
Binary integer 123, 134
Binary integers 367
Bit 71, 342, 383
Bit clear 383
Bit set 383
Boolean algebra 342
Buffer 179, 183, 184, 185, 383
BufFull 179
BufLen 179

I. Index 393
C 77
C->SB 258
Cache buffer 85, 92
CAL 320
CALC 266
Calling subroutines 84, 112
Card present in port 168
CARRY 76
Character 123, 129, 133, 359
Checksum 143, 144
CHECK_BAT 390
CHECK_BATI 390
CHK 232
CIRCLE 322
CLEAN 218
Clock 159, 176, 177
Clock Offset 176, 178
CLR 22
CLVAR 239
CMOS word 176, 177
Code 123, 153
COMA 177
Command line 175, 191
Command Line Stack 197
Comments 51
Comparing registers 84, 113
Comparisons 84, 113
Complex Numbers 123, 127, 359
CONFGGRAPH 391
Config. Object 143
Configuration Table 181
CONFTABCRC 389
Contrast 164, 165, 241
Control code 148, 149
Control instructions 84, 120
Converting Between Bases 343
CPU cycles 84
CRC 143, 148, 149, 164, 166, 265
CRC calculator 166
CRC value 149
CRCLM 265
CRDIR 33
CRNAME 238
CST 48
Current Directory 194
Current menu offset 201
Cycles 84
Cyclic Redundancy Control 148, 166
D 77, 175, 189, 210
D0 76, 77, 210
D07FMAP 389
D0TOS 390
D1 76, 77, 175, 186, 189, 210
D17FMAP 389
D17FMAP2 389
D1TOS 390
Data 179
Data pointer registers 73, 76
Decrement 83, 99
DEPTH 24
DER 288
Derivatives 59
Direct relative conditional 84, 111
Direct relative unconditional 84, 111, 112
Directories 16, 29, 123, 136
Disable keyboard 186
Disable system-halt 176
DISASM 243
Disassembler 384
Disassembling 70
DISINTR 390
DISPINCROB 390
Display 164, 165
DISPOFF 241, 390
DISPON 241, 390
DIV 276, 389
DIV5 388
Dividing by 16 83, 107
Dividing by 2 83, 106
DIVP 290
DROP 22, 26
DROP2 26
DROPN 26
DSP 312
DUP 25
DUP2 25
DUPN 25
E 266
EEPROM 53
EMPTATTN 390
EMPTKBUF 390
Empty name 196
End of RAM 201
Ending @ of free memory 186
Ending @ of temporary objects 186
Epilog 135, 139, 140, 152
EPROM 53
Error 177, 388
Error messages 146, 374
Error number 201, 204
Exchanging register contents 83, 94
Exchanging register fields 83
EXHR 388
EXponent 77, 78, 125, 126, 127, 128
External 157
External module missing 76
Factorial 2000
FAST 165, 242
Field pointer register 73, 77
Fields 77
Finding extrema 59
Flag 384
Flag registers 73, 76
Flags 191, 201, 202
Free memory 175, 189, 201
FREEMEM 389
FREEMEMQ 389
Garbage collector 188, 384
GARB_COLL 387
GASS 209, 215
Getting the program counter 84, 111
Global name 123, 154, 372
Global variables 41
Graphics object 123, 142, 188, 371
Graphs 59
GROB of the character under the Cursor 204
Hash Table 136, 143, 145
Hexadecimal 71, 343, 384
Hidden directory 196
Hidden memory 159
Hidden ROM 170, 224
High-level language 69
HOME 32, 33, 137, 194
HP28 73
HP71 73
HRPEEK 224
HST 76
I/O RAM 159, 160, 163
I/O registers 73
Icons 50
Immediate 83, 84, 86, 113
Increment 83, 97
Infrared 56, 177
Infrared receiver/transmitter 159
INPUT 73, 164
Input and output 83, 93
Input OK 169
Instruction Set 83
Instructions with no effect 84
Interrupt Backup 182
Interrupts 80, 164, 169, 182
IR 56
IR in mem. 164
IR input 164, 169
IR output 164, 170
IR7CONF 391
IRFCONF 391
JINGLE 318
Jumps 83, 110
KERMIT 55, 149
Key codes 184, 185, 186
Key state 183, 184, 186
Keyboard 15, 73, 179, 184, 185
Keyboard Buffer 184, 185, 186
KeyEnd 184, 186
KEYINBUFF 390
KeyStart 184, 186
LAGU 292
Large binary integer 134, 145, 147
LAST 197
Last menu offset 201
Last RPL Token 201
LAST Stack 197
LCD 384
Left Shift 164
Library 123, 136, 143
Library Data 123, 136
Library number 137, 143
Link Table 143, 147
Linked array 123, 132
LISP 35
List 123, 135
Lists 368
LOAD_REG 210, 387
Local name 123, 155, 372
Local variable 41, 191, 192
Logical AND 83, 102
Logical NOT 83, 104
Logical OR 83, 103
Long Complex Numbers 123, 128, 359
Long Real Numbers 123, 126, 358
Low-level language 69
M 77, 78
Machine language 69, 384
Machine speed 186
Making data access easier 48
Managing the Stack 22
Mantissa 77, 126, 127, 128, 132
Margin 164, 165
MASTER 304
MAZE 298
MEM 29
Memory 66, 384
Menu 16
Menu bar 172, 198
Menu bar bitmap 171
Menu bar height 171, 172
Menu bitmap 198
Menu bitmap address 186
Menu height 186
Menu Offsets 204
Menu trees 31
Menus 29, 198
MERGE 54
Message 146
Message Table 136, 143, 146
Index
Microprocessor 71
Mini editor screen prep. 176
Mini-editor 160
Mini-Editor Screen Preparation 178
Miscellaneous notes 79
MODU 266, 282
MODUL 316
Module pulled 76
MODUSEARCH 264
Moves 83, 86
Moving values 88
MP 76
mSOLVER 295
MULT 266, 283, 389
MULTA 388
Multiplying by 16 83, 107
Multiplying by 2 83, 105
MUSICML 314

NEXT 29
Next error to display 201
Next Object to be Executed 197
Nibble 342, 385
NOPs (instructions with no effect) 84, 120
Number of attached libraries 136, 201, 204
Numerical calculation 59

Object 385
Objects 35, 123, 354
ON-D 160
Other Objects 157
OUTPUT 164
Output Mask for the Keyboard Test 182
Output OK 169
OVER 22

P 78
PATH 33
PC 76
PCAR 291
PEEK 220, 385
Peripherals 159
PI 286, 287
PICK 24
Pixel 142
Plug-in card 179
Plug-in card ports 159
Plug-in card removed 177
Plug-In cards 53, 179, 181
PMAT 294
POKE 222, 385
POP C 388
POP_C_A 388
Port 179, 181
Port 0 194
Port 1 161, 179
Port 2 161, 179
Port information (HP48sx) 179, 181
POW 266, 282

PR40 311
PREVIOUS 29
Processor 385
Program 36, 123, 152
Programming Methods 36
Prolog 385
PROM 53
Pseudo operations 84, 120
PUSH_R0 388
PUSH_R0_R1 388

R->SB 258
R0 76
R1 76
R2 76
R3 76
R4 76
RABIP 318
RAM 53, 54, 175, 385
RAMSEARCH 263
Random number seed 201, 202
RASS 230
Reading and writing to memory 83, 92
Real number 123, 125
Real Numbers 358
Real/Complex array 123, 130
Recursion 42
Redefining keys 50
Register 385
Register direct 84, 111
Register indirect 84, 111
Registers 73
Registers used by the HP48 79
RENAME 319
Reserved 1 123
Reserved 1, 2, 3 and 4 151
Reserved 2 123
Reserved 3 123
Reserved 4 123
Reserved RAM 175
Restart 177
RES_ROOM 387
RES_STR 387
Return stack 76, 175, 189
Returning from subroutines 84, 112
REVERSE 236
Reverse Polish Lisp 35
Reverse Polish Notation 18
Right Margin 171, 172
Right Shift 164
ROLL 23
ROLLD 23
ROM 53, 159, 385
ROMRCL 259
ROMSEARCH 263
ROT 22
Rotating left (one nibble) 83, 108
Rotating right (one nibble) 83, 109
RPL 35
RPL Commands 345, 350
RS232c 54, 164, 179, 385
RS232c Input 164, 169, 170
RS232c Input Buffer 180
RS232c Interrupts 164, 169
RS232c Output 164, 169, 170
RS232c Speed 164, 168
RSTK 76

S 77, 78
Saturn 73
SAVE_REG 210, 387
Saving and Restoring (Rn and RSTK) 83, 90
SB 76
SB->B 258
SB->C 258
SB->R 258
Scratch registers 73, 76
Screen 16, 160
Screen bitmap 78, 171, 172, 186
Screen bitmap addr. 171, 172, 186
Screen GROBS 175
Service request 76
Sign 77, 126, 127, 128
Solving equations 59
Sound 74
Speaker 74
SQR 266, 283
SQUARE 308
SR 76
SSAG 229
ST 76
Stack 16, 175, 385
Stack size 190, 201, 202
Statistical functions 59
STATUS 76
Sticky bit 76
STO 32
STOP 388
Store 32
STR->A 260
String 123, 133
Strings 363
SUBS 266, 283
Subtraction 83, 100
SWAP 22
Symbol @ 66
Symbolic Calculations 59
SYSEVAL 240
System Binaries 123, 124, 355
System Flags 202

TAG 51, 141
Tagged object 123, 141
Taylor's Approximation 59
Temporary backup during interrupts 179
Temporary environment 175, 193
Temporary objects 175, 188
Time 61

Timer1 171, 173
Timer2 171
Transmitting 164
TRDN 387
TRUP 387

Understanding programs easier 51
Undo Stack 191, 192
Undo stack, local variables 175
Unit 140
Unit object 123, 140
Units 61, 370
UPDIR 33
Useful Routines 387
User Flags 203
User Stack 189
User variables 175
User-keys 196

V->A 289
VAL 288
"VAR" Menu 32
Variables and directory trees 41
VSYNC 171, 173

W 77, 78
Wide 78
Wide-P 78
Working registers 73, 77
WP 78
WSLOG 160, 176, 177

X 77, 78, 79
XLIB name 123, 156
XM 76
XS 77, 78

ZEROM 387

I. Index