An Introduction to HP 48 System RPL and Assembly Language Programming

James Donnelly

March 1, 2009

Acknowledgements

This book would not exist were it not for the team that developed the original HP 28. The tribute to their vision exists in backpacks, briefcases, and on desktops around the world.

Inspiration for the book came from many places, notably the traffic on comp.sys.hp48. Doug Cannon and Brian Maguire were the principals that helped get the project going and provided valuable examples and suggestions.

Seth Arnold, Lee Buck, Rick Grevelle, Wlodek A.C. Mier-Jedrzejowicz, Richard Nelson, Jeremy Smith, and others repeatedly provided encouragement when the going got tough and I thought about abandoning the project. Perhaps the most consistent motivation came from the not-infrequent posting on comp.sys.hp48 from new HP 48 owners asking for examples and tips for getting started.

Seth Arnold, Ted Beers, Lee Buck, Doug Cannon, Yuan Feng, Joseph Horn, Wlodek A.C. Mier-Jedrzejowicz, Brian Maguire, and Eric L. Vogel all contributed to the review process.

Hewlett-Packard provided permission to distribute copies of their HP 48 development tools on the disk that accompanies this book.

Immense credit goes to my wife Janet, who supported and encouraged this project, and thus was left alone for the many hours of writing, testing, debugging, and proofing.

Disclaimer

Contents

1 Introduction **11**

2 Getting Started **13**

2.1 Terminology 13

2.1.1 User-RPL vs. System-RPL vs. Assembler 13

2.1.2 Stack Diagrams 14

2.1.3 Object Notation 14

2.1.4 Fonts 15

2.2 Installing the HP Tools 15

2.3 Example Programs 16

2.4 Introducing System-RPL 16

2.4.1 A First Example 16

2.4.2 Creating the Example With the HP Tools 17

2.5 Introducing Assembly Language 18

2.6 Example File Structures 19

2.6.1 User-RPL Examples 20

2.6.2 System-RPL Examples 20

2.6.3 Assembly Examples 21

3 Basic Programming Tools **23**

3.1 Binary Integers 23

3.1.1 Internal Binary Integers in the HP 48 Display 23

3.1.2 Internal Binary Integers in System-RPL Source Code 23

3.1.3 Type Conversions 25

3.1.4 Internal Binary Integer Operations 26

3.2 Flags 30

3.2.1 Flag Conversions 30

3.2.2 Flag Utilities 31

3.3 Tests 31

3.3.1 Object Equality 32

3.3.2 Binary Integer Tests 32

3.3.3 Real Number Tests . . . 33
3.3.4 Extended Real Number Tests . . . 34
3.3.5 Complex Number Tests . . . 35
3.3.6 Advanced Topic: Missing Extended Real Test Objects . . . 35
3.3.7 Unit Object Tests . . . 36
3.3.8 Character String Tests . . . 37
3.3.9 Hex String Tests . . . 37
3.4 Program Flow Control . . . 37
3.4.1 Early Exits From a Secondary . . . 38
3.4.2 IF – THEN – ELSE Structures . . . 38
3.4.3 CASE Objects . . . 41
3.5 Loop Structures . . . 48
3.5.1 Definite Loops . . . 48
3.5.2 Indefinite Loops . . . 50
3.6 Runstream Operators . . . 51
3.7 Argument Validation . . . 52
3.7.1 Attributing Errors . . . 52
3.7.2 Number of Arguments . . . 53
3.7.3 Type Dispatching . . . 54
3.7.4 Object Type Tests . . . 57
3.8 Temporary Variables . . . 58
3.8.1 Using Named Temporary Variables . . . 60
3.8.2 Using Null-Named Temporary Variables . . . 60
3.8.3 Programming Hint for Temporary Variables . . . 62
3.8.4 Additional Temporary Variable Utilities . . . 63
3.9 Error Trapping . . . 64
3.9.1 Error Trapping Mechanics . . . 65
3.9.2 Generating an Error . . . 65
3.9.3 Handling an Error . . . 66
3.9.4 Additional Error Objects . . . 66
3.10 Stack Operations . . . 67
3.11 Control Structure Examples . . . 72
3.11.1 PLIST Example . . . 73
3.11.2 SEMI Example . . . 73
3.11.3 ticR Example . . . 74

4 Objects & Object Utilities **77**
4.1 Real & Extended Real Numbers . . . 77
4.1.1 Compiling Real Numbers . . . 77
4.1.2 Built-In Real Numbers . . . 78
4.1.3 Real Number Conversions . . . 79
4.1.4 Real Number Functions . . . 80
4.1.5 Extended Real Number Functions . . . 83
4.2 Complex Numbers . . . 85
4.2.1 Compiling Complex Numbers . . . 85
4.2.2 Complex Number Conversions . . . 86
4.2.3 Built-In Complex Numbers . . . 87
4.2.4 Complex Number Functions . . . 87
4.3 Arrays . . . 88
4.3.1 Compiling Arrays . . . 89
4.3.2 Array Utilities . . . 89
4.3.3 The MatrixWriter . . . 90
4.4 Tagged Objects . . . 90
4.5 Characters and Character Strings . . . 91
4.5.1 Built-In Character Objects . . . 91
4.5.2 Built-In String Objects . . . 92
4.5.3 String Manipulation Objects . . . 93
4.6 Hex Strings . . . 96
4.6.1 Hex String Conversions . . . 96
4.6.2 Wordsize Control . . . 97
4.6.3 Basic Hex String Utilities . . . 97
4.6.4 Hex String Math Utilities . . . 98
4.7 Composite Objects . . . 99
4.7.1 Building Composite Objects . . . 99
4.7.2 Finding the Number of Objects in a Composite Object . . . 100
4.7.3 Adding Objects to a Composite . . . 100
4.7.4 Decomposing Composite Objects . . . 101
4.7.5 Searching Composite Objects . . . 102
4.7.6 Detecting Embedded Objects . . . 102
4.8 Unit Objects . . . 102
4.8.1 Dimensional Consistency . . . 103
4.8.2 Building and Decomposing Unit Objects . . . 103
4.8.3 Unit Object Utilities . . . 103

5 Memory Utilities **107**
5.1 Name Objects . . . 108
5.2 User Variables . . . 108
5.3 Directory Utilities . . . 109
5.4 Temporary Memory . . . 110
5.4.1 Use of Temporary Memory . . . 110
5.4.2 Garbage Collection . . . 111
5.5 Memory Utilities . . . 112

6 Graphics, Text, and the LCD **113**
6.1 LCD Display Regions . . . 113
6.1.1 Status Area Control . . . 113
6.1.2 Stack Area Control . . . 114
6.1.3 Menu Area Control . . . 114
6.1.4 Combined Area Controls . . . 115
6.2 Basic Display Memory Principles . . . 115
6.2.1 The Current Display Grob . . . 115
6.2.2 The Stack Grob . . . 116
6.2.3 The Graphics Grob . . . 117
6.2.4 Verifying Display Grob Height . . . 117
6.2.5 Enlarging ABUFF or GBUFF . . . 117
6.2.6 Scrolling ABUFF or GBUFF . . . 118
6.2.7 The Menu Grob . . . 119
6.2.8 Display Pointer Examples . . . 119
6.3 Graphics Coordinates . . . 120
6.3.1 Subgrob Coordinates . . . 120
6.3.2 User Pixel Coordinate - Bint Conversion . . . 121
6.3.3 User-Unit to Pixel Conversion . . . 121
6.3.4 Accessing PPAR . . . 121
6.4 Displaying Text*PICT* . . . 122
6.4.1 Medium Font Display Objects . . . 122
6.4.2 Displaying Temporary Messages . . . 123
6.4.3 Large Font Display Objects . . . 124
6.5 Basic Grob Tools . . . 125
6.5.1 Creating Grobs . . . 125
6.5.2 Finding Grob Dimensions . . . 125
6.5.3 Extracting a Subgrob . . . 126
6.5.4 Inverting a Grob . . . 126
6.5.5 Combining Graphics Objects . . . 126
6.5.6 Clearing a Grob Region . . . 126

6.6 Drawing Tools . . . 127
6.6.1 Line Drawing . . . 127
6.6.2 Pixel Control . . . 127
6.7 Menu Grob Utilities . . . 128
6.8 Built-in Grobs . . . 128
6.9 Graphics Examples . . . 129
6.9.1 Drawing a Grid . . . 129
6.9.2 A Rocket Launch . . . 131

7 Keyboard Utilities 133
7.1 Key Buffer Utilities . . . 133
7.2 Checking The Keyboard While Running . . . 133
7.2.1 Detecting the [ON] Key . . . 134
7.2.2 Detecting Any Key . . . 134
7.3 Waiting For a Key . . . 136
7.4 Keycodes . . . 137
7.5 Repeating Keys . . . 138
7.6 InputLine . . . 139
7.6.1 Input Parameters . . . 140
7.6.2 InputLine Results . . . 141
7.6.3 InputLine Examples . . . 142

8 The Parameterized Outer Loop 145
8.1 Introducing ParOuterLoop Parameters . . . 145
8.2 Temporary Environments and the POL . . . 151
8.3 The Exit Object . . . 151
8.4 The Error Object . . . 151
8.5 Display Objects . . . 152
8.6 Hardkey Handlers . . . 152
8.6.1 Key and Plane Codes . . . 152
8.6.2 Hardkey Handler Structure . . . 153
8.7 Softkey Definitions . . . 157
8.7.1 Null Menu Keys . . . 158
8.7.2 Softkey Label Objects . . . 158
8.7.3 Softkey Action Object . . . 159
8.8 The POL Error Trap Object . . . 161
8.9 POL Utilities . . . 162
8.10 Menu Utilities . . . 164

9 Graphical User Interfaces **167**
9.1 Message Boxes 167
9.1.1 Message Box Parameters 168
9.1.2 Message Box Example 168
9.2 Equation Library Browser 169
9.2.1 Browser Parameters 170
9.2.2 Active Browser Keys 170
9.2.3 Browser Support Objects 171
9.2.4 Browser Example 171
9.3 Choose Boxes 173
9.3.1 Choose Box Styles 173
9.3.2 Choose Box Parameters 174
9.3.3 Choose Box Message Handler 175
9.3.4 Decompile Objects 177
9.3.5 Customizing Choose Box Menus 179
9.3.6 Choose Event Procedures 182
9.4 Input Forms 183
9.4.1 Input Form Parameters 184
9.4.2 Label Specifiers 185
9.4.3 Field Specifiers 185
9.4.4 Input Form DEFINEs for RPLCOMP 187
9.4.5 Specifying Object Types 188
9.4.6 Specifying Decompile Formats 189
9.4.7 Input Form Message Handlers 190
9.4.8 Input Form Data Access 190
9.4.9 Customizing Input Form Menus 191
9.4.10 ORBIT Example 193

10 Introducing Saturn **199**
10.1 The Saturn CPU 199
10.1.1 The Working and Scratch Registers 200
10.1.2 The Status Bits 201
10.1.3 Input and Output Registers 202
10.1.4 The Return Stack 202
10.1.5 Arithmetic Mode 202
10.1.6 The Pointer Register 202
10.2 Instruction Set Summary 202
10.2.1 Memory Access Instructions 203
10.2.2 Load Constant Instructions 203
10.2.3 P Register Instructions 204

10.2.4 Scratch Register Instructions . . . 204
10.2.5 Shift Instructions . . . 204
10.2.6 Logical Instructions . . . 205
10.2.7 Arithmetic Instructions . . . 205
10.2.8 Branching Instructions . . . 206
10.2.9 Test Instructions . . . 207
10.2.10 Register & Status Bit Instructions . . . 209
10.2.11 System Control Instructions . . . 210
10.2.12 Keyscan Instructions . . . 210
10.2.13 NOP Instructions . . . 210
10.2.14 Assembler Pseudo-Op Instructions . . . 210

11 Writing Your Own Code Objects **213**
11.1 Code Object Execution . . . 213
11.2 Stack Access . . . 214
11.2.1 Example: SWAP Two Objects . . . 215
11.2.2 Example: DROP Nine Objects . . . 216
11.3 Reading Assembly Language Entry Descriptions . . . 217
11.4 Saving and Restoring the RPL Pointers . . . 217
11.4.1 Example: Reversing Objects on the Stack . . . 217
11.4.2 Example: Clearing A Grob . . . 218
11.5 Stack Utilities . . . 219
11.5.1 Pop Utilities . . . 220
11.5.2 Push Utilities . . . 220
11.5.3 Examples: Indicated ABS . . . 223
11.6 Memory Utilities . . . 224
11.6.1 Allocating Memory . . . 224
11.6.2 Memory Move Utilities . . . 225
11.6.3 Display Memory Addresses . . . 227
11.7 Reporting Errors . . . 228
11.8 Checking Batteries . . . 228
11.9 Warmstart & Coldstart . . . 229
11.10 Tone Generation . . . 229
11.10.1 Steady Tones . . . 229
11.10.2 Rising and Falling Tones . . . 230
11.11 Keyboard Scanning . . . 232
11.11.1 Managing Interrupts . . . 232
11.11.2 Rapid Keyboard Scans . . . 233
11.11.3 Low Power Keyboard Scans . . . 235
11.12 The RVIEW Debugging Tool . . . 240
11.12.1 The RVIEW User Interface . . . 240
11.12.2 Using RVIEW . . . 241
11.13 The PONG Game . . . 241

A Messages 243

B Character Codes 249

C Flags 251

D Object Structures 255

D.1 Binary Integer . . . 255
D.2 Real Number . . . 255
D.3 Extended Real Number . . . 255
D.4 Complex Number . . . 255
D.5 Extended Complex Number . . . 256
D.6 Character . . . 256
D.7 String . . . 256
D.8 Hex String . . . 256
D.9 Arrays . . . 256
D.9.1 One-Dimension Array . . . 256
D.9.2 Two-Dimension Array . . . 256
D.9.3 Linked Array . . . 257
D.10 Name Objects . . . 257
D.10.1 Global Name . . . 257
D.10.2 Local Name . . . 257
D.10.3 XLIB Name . . . 257
D.11 Graphic Object . . . 257
D.12 Code Object . . . 257
D.13 Secondary . . . 257
D.14 Tagged . . . 258
D.15 List . . . 258
D.16 Symbolic . . . 258
D.17 Unit . . . 258
D.18 Library Data Objects . . . 258

Chapter 1

Introduction

The HP 48 calculator family is characterized in part by the availability of a wide variety of software products that address diverse interests, ranging from games to serious engineering applications. Some programs appear to run much faster than you would suspect possible if all your HP 48 programming experience was confined to standard programming from the keyboard. This book is designed to introduce some of the techniques used to create these programs.

The discussion and examples in this book have been drawn from the collective experience of the author and other contributors — each having a unique view of the HP 48. This book is an *introduction* to the HP 48 — we cannot and do not attempt to provide either complete documentation for every facet of the HP 48's internal resources *or* a complete theoretical description of the operating system. We do hope you will learn a few things, have some fun, and write some new programs for others to enjoy.

As with any book, we make some assumptions about the background of the reader. In particular, we assume the reader is familiar with all HP 48 object types and most basic HP 48 programming constructs. We recommend *The HP 48 Handbook*, by the author, as a good place to begin the study of User-RPL programming. The *Handbook* has lots of examples, and should get you started in good form. In particular, study pages 3-200.

Several tools exist that can be utilized to create programs using the HP 48's internal resources in ways not possible from the keyboard. The disk that comes with this book includes free copies of the tools provided by (but not supported by) Hewlett-Packard.

The chapters in this book are organized to provide a progression from fairly straightforward usage of some system resources in standard programs to complex application projects. However, this is *not* a novel with a plot that is linear through the book. For instance, some example programs use objects described later in the book. The book has been designed to act both as tutorial and reference, so you'll find yourself going back-and-forth from time to time.

Chapter 2

Getting Started

Any technical dialog is necessarily filled with terms that may confuse the reader new to the subject. We begin by defining some basic terms, introducing the tools, System-RPL, and assembler.

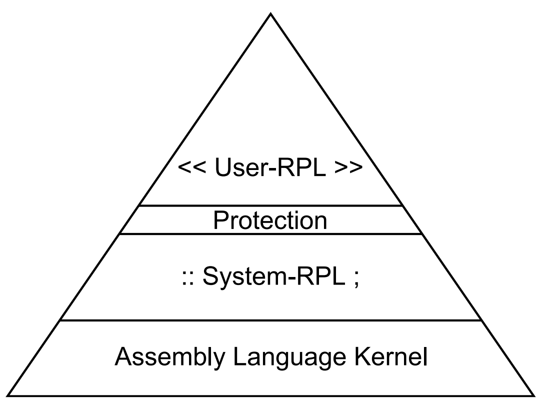
2.1 Terminology

The kernel of the HP 48 operating system/language known as RPL has been written in assembly language, and much of the functionality of in the HP 48 is implemented in what is sometimes called “System-RPL”. Programs entered from the keyboard of the HP 48 are written in what is sometimes called “User-RPL”.

Programs written in assembly language are often known as “code objects” (type 25) and can use all the resources in the HP 48. Unfortunately, the HP 48 has not been provided with a complete debugging environment for assembly language development. Consequently there have been fewer applications or games written in assembly language. This book will describe some techniques that can be applied to assembly language development projects.

2.1.1 User-RPL vs. System-RPL vs. Assembler

The illustration below shows the relationship between User-RPL, System-RPL, and the kernel of the HP 48.



Programs written in User-RPL and System-RPL share the same resources, stack, return stack, etc. The commands available in User-RPL represent a subset of the functionality available in System-RPL. The objects that can be used by System-RPL represent a subset of the HP 48 system.

There are three main distinctions between User-RPL and System-RPL:

- User-RPL commands have names that are recognized when you enter them into the command line, whereas System-RPL objects must be accessed via either the `SYSEVAL` command or specialized tools.

- User-RPL commands have extra code responsible for validating input arguments (and thus require a bit of extra execution time), whereas System-RPL objects usually have little or no error protection. This layer of protection insures that invalid input arguments do not result in undesirable behavior by underlying code.

- There are more resources available to programs written in System-RPL. These resources include access to portions of the HP 48 system objects, additional object types (notably internal binary integers), and additional control structures which may provide improved execution flow control.

Applications written in assembler have the greatest speed potential, the greatest access to system resources, and the most difficult development process. The penalties for errors in assembly are sometimes greater than for System-RPL, meaning that Memory Lost events are more likely. This should discourage only the faint-hearted, however.

2.1.2 Stack Diagrams

A stack diagram notation is used in this book which describes the type and order of objects supplied to a command or program and the type and order of results. In the case of an object that can be used in a System-RPL application, the description includes the name, address, and stack diagram as follows:

`NAME`		Address
	Input *Output*	
	$\text{Level}_3\ \text{Level}_2\ \text{Level}_1 \rightarrow \text{Level}_3\ \text{Level}_2\ \text{Level}_1$	
Related Flags: Flags which may affect the result		

Unless mentioned otherwise, all entries will work on all versions of the HP 48. Entries specific only to the G/GX series of calculators carry the "G/GX" mnemonic by the address. Some objects are accessed by rompointer (XLIB name). These entries are indicated by a user binary integer value for LIBEVAL (not always safe — including the case shown below) in the center of the top line and the XLIB notation at the top-right:

`DoMsgBox`	`#000B1h`	G/GX XLIB 177 0
Displays a message box with a graphics object		
	"message" #maxwidth #minwidth grob menuobject → TRUE	

2.1.3 Object Notation

Hewlett-Packard has adopted a series of symbols to represent different object types. Some of these symbols are listed below, along with their object type, an example of what the decompiled object type looks like in System-RPL, and what the object looks like as displayed on the stack.

Symbol	Type	Object	System-RPL Example	Stack Example
%	0	Real number	`% 1.2345`	1.2345
C%	1	Complex number	`C% 2.3 4.5`	(2.3,4.5)
$	2	String	`"ABC"`	"ABC"
arry	3	Real array	`ARRY [ % 1 % 2 % 3 ]`	[1 2 3]
arry	4	Complex array	`ARRY [ C% 1 2 C% 3 4 ]`	[(1,2) (3,4)]
{ }	5	List	`{ "ABC" % 1.5 }`	{ "ABC" 1.5 }
id	6	Global name	`ID X`	'X'
lam	7	Local name	`LAM y`	'y'
::	8	Secondary object (program)	`:: x« id A %2 x+ x» ;`	« A 2 + »
symb	9	Algebraic	`DOSYMB ID X %2 x^ ;`	'X^2'
hxs	10	User Binary integer	`HXS 10 7F00000000000000`	# 247d
grob	11	Graphics object	`GROB E 2000080000ABCD`	Graphic 8 × 2
tagged	12	Tagged object	`TAG Dist % 34.45`	Dist: 34.45
symb	13	Unit object	`DOEXT ... ;`	32_ft/s^2
romptr	14	XLIB name	`ROMPTR domain`	XLIB 766 1
#	20	Internal binary integer	`247`	<247d>
%%	21	Extended real number	`%% 1.23456789012345`	Long Real
C%%	22	Extended complex number	`C%% 1.234 5.678`	Long Complex
lnkarry	23	Linked array	`LNKARRY [ % 1 % 2 % 3 ]`	Linked Array
chr	24	Character object	`CHR A`	Character
code	25	Code object	`CODE ... ENDCODE`	Code

Objects are composed of a *prologue* and a *body*. An object prologue indicates the type of object, and the body contains the information of interest. Some objects, like strings, have a length field after the prologue that indicates the size of the object. Objects are also classified as being *atomic* or *composite*. An atomic object is a single object, like a real number. The body of a composite object, like a list, consists of one or more objects. For details about individual objects, see the appendix *Object Structures* on page 255.

2.1.4 Fonts

A font convention has been adopted to help distinguish between text, source code, and comments. The fonts are used as follows:

« 1.23 + » The dot matrix font is used for User-RPL and text displayed in the HP 48 LCD.

`:: % 1.23 %+ ;` The Courier font is used for System-RPL or assembler source code.

Validate arguments An italic font is used for comments

2.2 Installing the HP Tools

Hewlett-Packard has graciously permitted the distribution of their tools on the disk that comes with this book. There are three basic steps to the installation of the HP tools:

1. Copy the .EXE files to a directory in your path, typically a \BIN directory. Then copy the file ENTRIES.O, and the SASM.OPC file from the TOOLS directory to a convenient directory on your hard disk. On many systems, this would be a \INCLUDE directory.

The next two steps involve checking the \AUTOEXEC.BAT file on your PC:

2. Make sure that the PATH variable includes the directory containing the tools from step 1.
3. Add the following line to your AUTOEXEC.BAT file: SET SASM_LIB= \INCLUDE. This tells the SASM assembler where the SASM.OPC file is located. If you place SASM.OPC in a directory other than \INCLUDE, make sure this line refers to the proper directory.

When these three steps have been completed, reboot your PC and you're ready to go. The examples in this book will assume that the files mentioned in step 1 above are in the \INCLUDE directory of your PC.

It is beyond the scope of this book to describe the details of the HP tools — you may wish to refer to the HP documentation on the disk for details about the tools.

2.3 Example Programs

There are three directories of example programs. Each example program comes with a DOS .BAT file that compiles a working copy of the example program, ready to download to your HP 48. Checksums and sizes are also provided to help confirm that an example program is properly installed.

> **Note:** Many example programs contain error checking, but most examples of code objects do not. You should always back up your calculator before experimenting with example programs or changes to example programs.

2.4 Introducing System-RPL

As mentioned before, System-RPL programming is a superset of the process used to create programs in User-RPL. The basic resources are the same, but System-RPL has its own notation and options not available in User-RPL.

2.4.1 A First Example

We begin by comparing two objects that compute the length of the hypotenuse of a right triangle — one written in User-RPL and the other written in System-RPL. The User-RPL example is called a program, but it's common in the world of System-RPL to use the term secondary for the example shown on the right.

User-RPL		**System-RPL**	
$Side_1$ $Side_2$ $\rightarrow$ $Side_3$		% %′ → %″	
27.5 Bytes		20 Bytes	HYPOT.S
«	*Start of program*	::	*Start of secondary*
`DUP * SWAP DUP *`	*Square both sides*	`DUP %* SWAPDUP %*`	*Square both sides*
`+`	*Add the squares*	`%+`	*Add the squares*
`√`	*Take the square root*	`%SQRT`	*Take the square root*
»	*End of program*	;	*End of secondary*

Note the differences between the two:

- Delimiters for a User-RPL program and a secondary written in System-RPL are different. Secondaries begin with `::` (called DOCOL), and finish with `;` (called SEMI).
- User-RPL programs are *self quoting* — they place themselves on the stack until explicitly executed — and secondaries are executed. See *Program Flow Control* on page 37 for more about this difference.
- We could have used `SQ` to square each side in the User-RPL example, but the actual code for the user command SQ (in the case of a real number) is `:: DUP %* ;` so we have used `DUP *` in place of `SQ`.
- The DUP used in the secondary is not the same as the User-RPL `DUP`. The User-RPL `DUP` checks the stack to make sure that at least one object is on the stack before duplicating it. The System-RPL `DUP` assumes that there is at least one object on the stack, and duplicates the object with no checks at all.
- In User-RPL, `*` encapsulates every possible multiplication operation. The System-RPL example uses `%*`, which multiplies two reals, and makes no argument checks. This is the object that is ultimately executed by the User-RPL `*` when it is asked to multiply two real numbers. Thus the System-RPL example avoids the time required to determine which multiply routine to use. The same logic applies to the use of `%+` and `%SQRT`.

- The System-RPL example is smaller for two reasons. First, the example uses `SWAPDUP`, which combines the operations of SWAP and DUP into one efficient piece of machine language. There are many such objects available through System-RPL that combine common operations into one operation. The use of `SWAPDUP` also saves space — this makes the System-RPL example 2.5 bytes shorter than it would have been if `SWAP` and `DUP` were used individually. The System-RPL example is also smaller because it lacks the `«»` delimiters found in the User-RPL program. The User-RPL program when decomposed actually contain `::` and `;` around the outer program delimiters, so internally the program actually looks like `:: « DUP * SWAP DUP * + √ » ;` . When a User-RPL program is displayed the `::` and `;` are suppressed.

- One hazard of using the System-RPL example to find the length of a hypotenuse is that there is no argument validation. If you're sure that only real numbers will be present on the stack when the secondary is executed, no problems should result. Invalid arguments supplied to the User-RPL program will generate a `Bad Argument Type` error; invalid arguments supplied to the System-RPL secondary will have unpredictable consequences, ranging from meaningless results to the loss of memory.

- Another consequence of the lack of argument validation is that the program does not clear the system RAM location that attributes the source of an error. If an error were to occur, it would be attributed to the last command that generated an error, which does no actual harm but is quite misleading.

- The System-RPL example will run faster than the User-RPL program, because all the argument checking code has been bypassed. In this example the speed difference is minor, but in future examples you'll begin to see where major speed improvements can be found.

The System-RPL example shown above has been written for maximum efficiency at the expense of argument validation. That may be appropriate for secondaries embedded in larger applications, but it is not recommended for general use when an inexperienced user might supply invalid input data. Later in the book we will show a technique for validating the arguments.

We now illustrate the process of compiling the System-RPL example using the HP tools on a PC.

2.4.2 Creating the Example With the HP Tools

To prepare the example, you will compile, assemble, and load the code using a source code file, a loader control file, and a batch file to automate the process. The input files HYPOT.S, HYPOT.M, and the batch file HYPOT.BAT are listed below:

`HYPOT.S`	*This is the source code file for the program.*
`ASSEMBLE`	*A pseudo-op that tells the compiler to pass the next output to SASM*
`    NIBASC /HPHP48-A/`	*This is a download header for binary transmission to the HP 48*
`RPL`	*A pseudo-op that tells the compiler to compile the source that follows*
`::`	*The beginning of the source code*
`    DUP %* SWAPDUP %*`	
`    %+`	
`    %SQRT`	
`;`	

`HYPOT.M`	*This is the loader control file that controls the execution of the loader SLOAD.*
`TITLE Hypotenuse`	*This is an optional title that will appear in the .LR output file*
`OUTPUT HYPOT`	*Instructs SLOAD to put the final output in the file HYPOT*
`LLIST HYPOT.LR`	*Instructs SLOAD to put listing information and errors in HYPOT.LR*
`SUPPRESS XREF`	*Suppresses a cross reference listing that would appear in HYPOT.LR*
`SEARCH \INCLUDE\ENTRIES.O`	*The reference to the addresses in ENTRIES.O*
`REL HYPOT.O`	*Specifies which file to load*
`END`	

`HYPOT.BAT`	*This is a batch file that encapsulates the entire process.*
`RPLCOMP HYPOT.S HYPOT.A`	*Invokes RPLCOMP, generates the SASM source file HYPOT.A*
`SASM HYPOT.A`	*Assembles HYPOT.A, generates HYPOT.L and HYPOT.O*
`SLOAD -H HYPOT.M`	*Invokes SLOAD using the control file HYPOT.M, generates HYPOT*

The file HYPOT.BAT encapsulates the entire process into a single batch file, so you have only one command to issue at the PC keyboard. Run HYPOT.BAT, which issues the commands to compile the .S source file, assemble the resulting .A file, and resolve the entry points with the .M file. Check HYPOT.L to make sure there were no compile or assembly errors.

Now examine the file HYPOT.LR. You should see something resembling the listing below:

HYPOT.LR

```
Saturn Loader, Ver.  %I%, %G%
Output Module:
Module=HYPOT
Start=00000 End=00037 Length=00038 Symbols=2293 References= 8
  Date=Sat Apr 22 14:20:28 1995 Title= Hypotenuse


Source modules:
Module=\INCLUDE\ENTRIES.O
  Start=00000 Module Contains No Code
  Date=Fri Apr 21 21:35:29 1995 Title=Supported ROM Entry Points
Fri Apr 21 21:35:29 1995

Module=HYPOT.O
  Start=00000 End=00037 Length=00038
Date=Sat Apr 22 14:20:28 1995 Title=
Sat Apr 22 14:20:28 1995

/SLOAD: End of Saturn Loader Execution
```

If an unresolved reference appears at the end of a .LR file, you most likely have specified an entry that is not in the file ENTRIES.O. Make sure that you have spelled the name correctly, which is the usual source of these errors.

To try out the System-RPL example, download the file HYPOT into your HP 48 and try it out with real numbers for input. Remember, the error checking that protected you is now gone. The section *Argument Validation* on page 52 in the chapter *Basic Programming Tools* shows how you can design your own argument validation routines.

2.5 Introducing Assembly Language

To introduce assembly language, we begin with one of the smallest possible examples — the HP 48's equivalent of "Hello World" in C programming. This program will return to the stack the address of the object in level 1 expressed as an internal binary integer. The HP 48 stack is merely a stack of 20-bit address pointers to objects residing in memory. The program copies the address into a CPU register, then branches to a routine that returns the address expressed as an internal binary integer.

To prepare the example, you will assemble and load the code using a source code file, a loader control file, and a batch file to automate the process. The input files ADDR.A, ADDR.M, and ADDR.BAT are listed below:

ADDR.A	*This is the source code file for the program.*
`    NIBASC \HPHP48-A\`	*This is a download header for binary transmission to the HP 48*
`    CON(5) =DOCODE`	*This is the prologue for a code object*
`    REL(5) end`	*The length field — indicates the size of the code object*
`    GOSBVL =SAVPTR`	*Saves the RPL pointers*
`    A=DAT1 A`	*Reads the pointer from stack level 1 into the A field of register A*
`    GOVLNG =PUSH#ALOOP`	*Pushes the A field of register A as an internal binary integer,*
`END`	*restores the RPL pointers, and returns to RPL*

ADDR.M	*This is the loader control file that controls the execution of the loader SLOAD.*
OUTPUT ADDR	*Instructs SLOAD to put the final output in the file ADDR*
LLIST ADDR.LR	*Instructs SLOAD to put listing information and errors in ADDR.LR*
SUPPRESS XREF	*Suppresses a cross reference listing that would appear in ADDR.LR*
SEARCH \INCLUDE\ENTRIES.O	*The reference to the addresses in ENTRIES.O*
REL ADDR.O	*Specifies which file to load*
END	

ADDR.BAT	*This is a batch file that encapsulates the entire process.*
SASM ADDR.A	*Assembles ADDR.A, generates ADDR.L and ADDR.O*
SLOAD -H ADDR.M	*Invokes SLOAD using the control file ADDR.M, generates ADDR*

The file ADDR.BAT encapsulates the entire process into a single batch file, so you have only one command to issue at the PC keyboard. Run ADDR.BAT, then examine the file ADDR.LR. You should see something resembling the listing below:

ADDR.LR

```
Saturn Loader, Ver.  %I%, %G%

Output Module:
Module=ADDR
Start=00000 End=0002A Length=0002B Symbols=2293 References= 3
  Date=Sat Apr 22 14:21:13 1995 Title=


Source modules:
Module=\INCLUDE\ENTRIES.O
  Start=00000 Module Contains No Code
  Date=Fri Apr 21 21:35:29 1995 Title=Supported ROM Entry Points
Fri Apr 21 21:35:29 1995

Module=ADDR.O
  Start=00000 End=0002A Length=0002B
Date=Sat Apr 22 14:21:13 1995 Title=
Sat Apr 22 14:21:13 1995

/SLOAD: End of Saturn Loader Execution
```

If an unresolved reference appears at the end of a .LR file, you most likely have specified an entry that is not in the file ENTRIES.O. Make sure that you have spelled the name correctly, which is the usual source of these errors. You may also want to check the .L file after assembly to check for compilation or assembly errors.

To try out the example, download the file ADDR into your HP 48 and try it out with the real number 1 on the stack. If the HP 48 is in HEX mode, you should see the internal binary integer `<2A2C9h>` on the stack, which is the address of the built-in constant 1. Notice also that if you recall ADDR to the stack, the program appears as `Code`. A code object (type 25) cannot be decompiled directly on the HP 48, but the Jazz tools (available on various FTP sites) can be used for assembly language development directly on the HP 48.

2.6 Example File Structures

The disk supplied with this book contains a directory named EXAMPLES. There are six subdirectories:

HPTOOLS	Contains the HP tools
USERRPL	Contains example programs written in User-RPL
SYSRPL	Contains example programs written in System-RPL
ASSEMBLY	Contains example programs written in assembly language

`RVIEW`	Contains the RVIEW register viewer
`PONG`	Contains the assembly language PONG game

2.6.1 User-RPL Examples

The User-RPL example programs are ready to download to the HP 48 in ASCII format. These files are named with a `.RPL` extension.

2.6.2 System-RPL Examples

The System-RPL examples consist of a *source file*, a *loader control file*, and a DOS batch file which will build the example program. A naming convention is used for these files. To illustrate the naming convention, consider the example program CASE1 described in *Case Objects* on page 41.

The input files are:

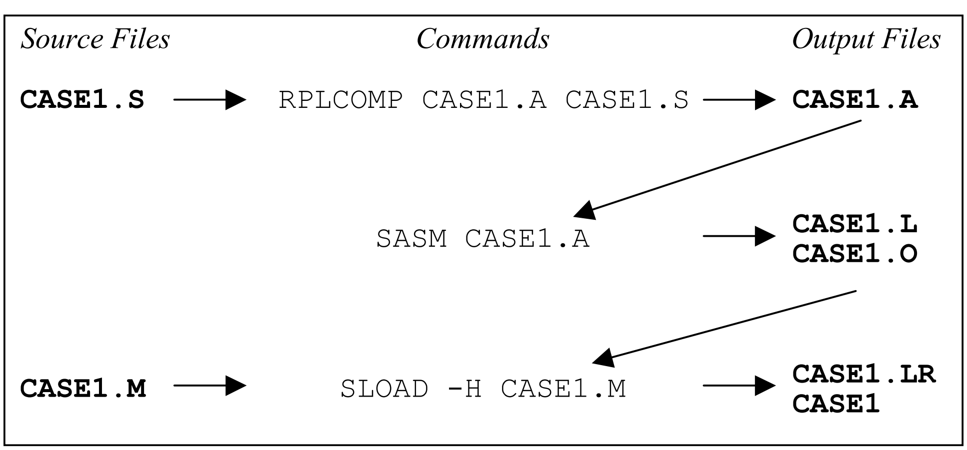
`CASE1.S`	The System-RPL source file
`CASE1.M`	The loader control file
`CASE1.BAT`	The DOS batch file

To compile and load the CASE1 example, just type CASE1 at the PCs command line, and the CASE1.BAT batch file will issue the commands to compile and load the example.

The output files are (in order of their creation):

`CASE1.A`	The assembler source generated by the RPL compiler RPLCOMP from CASE1.S
`CASE1.L`	The assembler listing file generated by the assembler SASM
`CASE1.O`	The object file generated by the SASM
`CASE1.LR`	The listing output from the loader SLOAD
`CASE1`	The example ready to download to the HP 48

The following diagram illustrates this process.



2.6.3 Assembly Examples

Like the System-RPL examples, the assembly language examples consist of a *source file*, a *loader control file*, and a DOS batch file which will build the example program. A similar naming convention is used for these files. To illustrate the naming convention, consider the example program SWP described in *Writing Your Own Code Objects* on page 215.

The input files are:

`SWP.A`	The assembler source file
`SWP.M`	The loader control file
`SWP.BAT`	The DOS batch file

To compile and load the SWP example, just type SWP at the PC's command line, and the SWP.BAT batch file will issue the commands to assemble and load the example.

The output files are (in order of their creation):

`SWP.L`	The assembler listing file generated by the assembler SASM
`SWP.O`	The object file generated by the SASM
`SWP.LR`	The listing output from the loader SLOAD
`SWP`	The example ready to download to the HP 48

Chapter 3

Basic Programming Tools

Programs written in System-RPL have a rich set of options for execution control, local variable use, and argument validation. This chapter will introduce some of the basic tools and program structures that you will use many times. There are a number of object types used by System-RPL objects which are not available in the User-RPL programming environment. The most prevalent of these are internal binary integers and the system flags `TRUE` and `FALSE`. These will be introduced first in the sections *Binary Integers* and *Flags*, because they're used everywhere else. The section *Tests* describes objects that perform various kinds of tests. These sections are followed by an introduction to some execution control constructs in the section *Program Flow Control*. When you are designing a System-RPL program, you should evaluate the precautions necessary to prevent the unwary user from getting unexpected results from invalid or missing input data. The section *Argument Validation* will describe the tools available for these tasks. The section *Temporary Variables* will describe the use of temporary environments, which are more flexible than the local variables found in User-RPL programs.

3.1 Binary Integers

Internal binary integers (sometimes nicknamed *bints*) are unsigned 20-bit quantities that are useful for many functions. These integers differ from user binary integers, which are actually stored internally as hex strings. To avoid confusion, this book will use the terms *user binary integer* and *internal binary integer* (or *bint*).

3.1.1 Internal Binary Integers in the HP 48 Display

While user binary integers (object type 10) are displayed with a leading `#` character, internal binary integers are displayed within `<>` symbols. A trailing character indicates the base display mode. For instance, if the base mode of the HP 48 is binary, then the internal binary integer 5 would be displayed as `<101b>`.

Internal binary integers live in the range $0 \leq n \leq$ FFFFF. If you subtract <1h> from <0h>, you get <FFFFFh> (decimal 1048575). No overflow or underflow indications are available.

3.1.2 Internal Binary Integers in System-RPL Source Code

The bad news is that in the world of System-RPL programming, the symbol # is used to denote internal binary integers, and the symbol `hxs` is used to denote User-RPL binary integers. Thus, when you see an object with a # in the name, the object probably works with internal binary integers. For instance, the object #+ adds two internal binary integers, returning an internal binary integer as the result.

The RPL compiler allows two notations for specifying internal binary integers. If the quantity is prefixed with the symbol #, then hex digits are expected. If no prefix character is present, the digits are interpreted as decimal values. Some commonly used bints (internal binary integers) are built into the HP 48, and can be accessed by name, saving 2.5 bytes from the 5 bytes taken by a compiled bint. The following secondary returns the same value three times:

```
::
    32          The decimal value 32 expressed as a bint
    # 20        The hex number 20h expressed as a bint
    THIRTYTWO   A pointer to the internal bint 32.
;
```

When the code listed above is compiled with RPLCOMP.EXE, the first two instances generate 5 bytes of code (values compiled as bint objects) and the third example generates 2.5 bytes (a pointer to a built-in bint):

```
CON(5) =DOCOL       The start of the secondary (::)
CON(5) =DOBINT      The prologue of an internal binary integer
CON(5) 32           The value of the bint
CON(5) =DOBINT      The prologue of an internal binary integer
CON(5) #20          The hex digits for the value 32
CON(5) =THIRTYTWO   The pointer to the built-in value of 32
CON(5) =SEMI        The end of the secondary (;)
```

Built-in Internal Binary Integers. The following objects put built-in internal binary integers on the stack:

Object	Stack Output	Address	Object	Stack Output	Address
MINUSONE	<FFFFFh>	#6509Eh	FORTYTHREE	<43d>	#0419Dh
ZERO	<0d>	#03FEFh	FORTYFOUR	<44d>	#64B12h
ONE	<1d>	#03FF9h	FORTYFIVE	<45d>	#64B1Ch
TWO	<2d>	#04003h	FORTYSIX	<46d>	#64B26h
THREE	<3d>	#0400Dh	FORTYSEVEN	<47d>	#64B30h
FOUR	<4d>	#04017h	FORTYEIGHT	<48d>	#64B3Ah
FIVE	<5d>	#04021h	FORTYNINE	<49d>	#64B44h
SIX	<6d>	#0402Bh	FIFTY	<50d>	#64B4Eh
SEVEN	<7d>	#04035h	FIFTYONE	<51d>	#64B58h
EIGHT	<8d>	#0403Fh	FIFTYTWO	<52d>	#64B62h
NINE	<9d>	#04049h	FIFTYTHREE	<53d>	#64B6Ch
TEN	<10d>	#04053h	FIFTYFOUR	<54d>	#64B76h
ELEVEN	<11d>	#0405Dh	FIFTYFIVE	<55d>	#64B80h
TWELVE	<12d>	#04067h	FIFTYSIX	<56d>	#64B8Ah
THIRTEEN	<13d>	#04071h	FIFTYSEVEN	<57d>	#64B94h
FOURTEEN	<14d>	#0407Bh	FIFTYEIGHT	<58d>	#64B9Eh
FIFTEEN	<15d>	#04085h	FIFTYNINE	<59d>	#64B8Ah
SIXTEEN	<16d>	#0408Fh	SIXTY	<60d>	#64BB2h
SEVENTEEN	<17d>	#04099h	SIXTYONE	<61d>	#64BBCh
EIGHTEEN	<18d>	#040A3h	SIXTYTWO	<62d>	#64BC6h
NINETEEN	<19d>	#040ADh	SIXTYTHREE	<63d>	#64BD0h
TWENTY	<20d>	#040B7h	SIXTYFOUR	<64d>	#64BDAh
TWENTYONE	<21d>	#040C1h	SIXTYEIGHT	<68d>	#64C02h
TWENTYTWO	<22d>	#040CBh	SEVENTY	<70d>	#64C16h
TWENTYTHREE	<23d>	#040D5h	SEVENTYFOUR	<74d>	#64C20h
TWENTYFOUR	<24d>	#040DFh	SEVENTYNINE	<79d>	#64C2Ah
TWENTYFIVE	<25d>	#040E9h	EIGHTY	<80d>	#64C34h
TWENTYSIX	<26d>	#040F3h	EIGHTYONE	<81d>	#64C3Eh
TWENTYSEVEN	<27d>	#040FDh	ONEHUNDRED	<100d>	#64CACh
TWENTYEIGHT	<28d>	#04107h	BINT_131d	<131d>	#64D24h
TWENTYNINE	<29d>	#04111h	BINT255d	<255d>	#64E28h
THIRTY	<30d>	#0411Bh	ZEROZERO	<0d> <0d>	#641FCh
THIRTYONE	<31d>	#04125h	ZEROZEROZERO	<0d> <0d> <0d>	#64309h
THIRTYTWO	<32d>	#0412Fh	ZEROZEROONE	<0d> <0d> <1d>	#6431Dh
THIRTYTHREE	<33d>	#04139h	ZEROZEROTWO	<0d> <0d> <2d>	#64331h
THIRTYFOUR	<34d>	#04143h	ONEONE	<1d> <1d>	#63AC4h
THIRTYFIVE	<35d>	#0414Dh	#FIVE#FOUR	<5d> <4d>	#642E3h
THIRTYSIX	<36d>	#04157h	#ONE#2	<1d> <27d>	#6428Ah
THIRTYSEVEN	<37d>	#04161h	#THREE#FOUR	<3d> <4d>	#642D1h
THIRTYEIGHT	<38d>	#0416Bh	#TWO#FOUR	<2d> <4d>	#642BFh
THIRTYNINE	<39d>	#04175h	#TWO#ONE	<2d> <1d>	#6429Dh
FORTY	<40d>	#0417Fh	#TWO#TWO	<2d> <2d>	#642AFh
FORTYONE	<41d>	#04189h	#ZERO#ONE	<0d> <1d>	#64209h
FORTYTWO	<42d>	#04193h	#ZERO#SEVEN	<0d> <7d>	#6427Ah

Other objects that put binary integers on the stack are listed under *Type Dispatching* on page 54.

3.1.3 Type Conversions

The objects COERCE and UNCOERCE convert between internal binary integers and real numbers. The objects COERCE2 and UNCOERCE2 convert two numbers. The stack diagrams for these objects are:

COERCE	#18CEAh
Converts a real number into an internal binary integer	
% → #	

COERCE2	#194F7h
Converts two real numbers into internal binary integers	
%x %y → #x #y	

UNCOERCE	#18DBFh
Converts an internal binary integer into a real number	
# → %	

UNCOERCE2	#1950Bh
Converts two internal binary integers into real numbers	
#x #y → %x %y	

Notice in these stack diagrams that we're using the shorthand mentioned before — % refers to real numbers and # refers to internal binary integers. Real numbers less than zero convert to <0>, values greater than 1048575 convert to <FFFFFh>, fractional parts <.5 round to the next lowest integer, and fractional parts $\geq$.5 round to the next highest integer.

3.1.4 Internal Binary Integer Operations

The following System-RPL objects operate on a single internal binary integer (bint):

Object	Description	Address
#1+	Adds 1 to a bint	#03DEFh
#1-	Subtracts 1 from a bint	#03E0Eh
#2+	Adds 2 to a bint	#03E2Dh
#2-	Subtracts 2 from a bint	#03E4Eh
#2*	Multiplies a bint by 2	#03E6Fh
#2/	Returns FLOOR(bint/2)	#03E8Eh
#3+	Adds 3 to a bint	#6256Ah
#3-	Subtracts 3 from a bint	#625FAh
#4+	Adds 4 to a bint	#6257Ah
#4-	Subtracts 4 from a bint	#6260Ah
#5+	Adds 5 to a bint	#6258Ah
#5-	Subtracts 5 from a bint	#6261Ah
#8+	Adds 8 to a bint	#625BAh
#8*	Multiplies a bint by 8	#62674h
#10+	Adds 10 to a bint	#625DAh
#10*	Multiplies a bint by 10	#6264Eh
#12+	Adds 12 to a bint	#625EAh

The following System-RPL objects operate on two internal binary integers:

#*	#03EC2h
Multiplies two bints	
#x #y → #x*y	

#+	#03DBCh
Adds two bints	
#x #y → #x+y	

#-	#03DE0h
Subtracts #y from #x	
#x #y → #x-y	

#/	#03EF7h
Divides #x by #y, returns remainder and quotient	
#x #y → #remainder #quotient	

#+-1	#63808h
Adds two bints, then subtracts 1 from the result	
#x #y → #x+y-1	

`#-#2/`		#624FBh
Subtracts #y from #x, divides the result by two, and returns the quotient		
#x #y	→	(#x-#y)/2

`#-+1`		#637CCh
Subtracts #y from #x, then adds 1		
#x #y	→	#x-#y+1

The following System-RPL objects combine stack operations (see *Stack Operations* on page 67) with binary integer numbers or arithmetic functions. They are quite useful for reducing the size of a program.

`2DROP00`		#6254Eh
Drops ob_1 and ob_2, then returns 0 0		
ob_2 ob_1	→	#0 #0

`2DUP#+`		#63704h
Duplicates #x and #y, then adds them		
#x #y	→	#x #y #x+y

`3PICK#+`		#63740h
Copies #x in level 3, then adds to #y		
#x ob #y	→	#x ob #x+y

`4PICK#+`		#63754h
Copies #x in level 4, then adds to #y		
#x ob_2 ob_1 #y	→	#x ob_2 ob_1 #x+y

`4PICK#+SWAP`		#62DE5h
Copies #x in level 4, adds to #y, then does SWAP		
#x ob_2 ob_1 #y	→	#x ob_2 #x+y ob_1

`#+DUP`		#627D5h
Adds #x and #y, then duplicates the result		
#x #y	→	#x+y #x+y

`#+OVER`		#63051h
Adds #x and #y, then copies object in level 2		
ob #x #y	→	ob #x+y ob

`#+ROLL`		#612DEh
Adds #x and #y, then does ROLL		
ob_{x+y} ... ob_1 #x #y	→	ob_{x+y-1} ... ob_1 ob_{x+y}

`#+SWAP`		#62DFEh
Adds #x to #y, then does SWAP		
ob #x #y	→	#x+y ob

`#-SWAP`		#62E12h
Subtracts #y from #x, then does SWAP		
ob #x #y	→	#x-y ob

`#-UNROLL`		#6132Ch
Subtracts #y from #x, then does UNROLL		
ob_{x-y} ... ob_1 #x #y	→	$ob_1 ob_{x-y}$ ... ob_2

`#1+DUP`		#62809h
Adds 1 to #x, then duplicates result		
#x	→	#x+1 #x+1

`#1+NDROP`		#62F75h
Drops #n+1 objects from the stack		
ob_{n+1} ... ob_1 #n	→	#

`#1+PICK`		#61172h
Copies the object in stack level #n+1		
ob_{n+1} ... ob_1 #n	→	ob_{n+1} ... ob_1 ob_{n+1}

`#1+ROLL`		#612F3h
Adds 1 to #x, then does ROLL		
ob_{x+1}... ob_1 #x	→	ob_x ... ob_1 ob_{x+1}

`#1+ROT` #1DABBh
Adds 1 to #x, then does ROT
$ob_2 ob_1$ #x $\rightarrow$ ob_1 #x+1 ob_2

`#1+SWAP` #62E26h
Adds 1 to #x, then does SWAP
ob #x $\rightarrow$ #x+1 ob

`#1+UNROLL` #61353h
Adds 1 to #n, then does UNROLL
$ob_{n+1} \ldots ob_1$ #n $\rightarrow$ $ob_1 ob_{n+1} \ldots ob_2$

`#1-1SWAP` #62E4Eh
Subtracts 1 from #x, then SWAPs #1 into level 2
#x #1 #x-1

`#1-DUP` #6281Ah
Subtracts 1 from #x, then duplicates the result
#x $\rightarrow$ #x-1 #x-1

`#1-ROT` #62F09h
Subtracts 1 from #x, then does ROT
$ob_2 ob_1$ #x $\rightarrow$ ob_1 #x-1 ob_2

`#1-SWAP` #5E4A9h
Subtracts 1 from #x, then does SWAP
ob #x $\rightarrow$ #x-1 ob

`#1-UNROT` #28558h
Subtracts 1 from #x, then does UNROT
$ob_2 ob_1$ #x $\rightarrow$ #x-1 ob_2 ob_1

`#2+PICK` #611BEh
Adds 2 to #n, then does PICK
$ob_{n+2} \ldots ob_1$ #n $\rightarrow$ $ob_{n+2} \ldots ob_1$ ob_{n+2}

`#2+ROLL` #61318h
Adds 2 to #n, then does ROLL
$ob_{n+2} \ldots ob_1$ #n $\rightarrow$ $ob_{n+1} \ldots ob_1$ ob_{n+2}

`#2+UNROLL` #61365h
Adds 2 to #n, then does UNROLL
$ob_{n+2} \ldots ob_1$ #n $\rightarrow$ $ob_1 ob_{n+2} \ldots ob_2$

`#3+PICK` #611D2h
Adds 3 to #n, then does PICK
$ob_{n+3} \ldots ob_1$ #n $\rightarrow$ $ob_{n+3} \ldots ob_1$ ob_{n+3}

`#4+PICK` #611E1h
Adds 4 to #n, then does PICK
$ob_{n+4} \ldots ob_1$ #n $\rightarrow$ $ob_{n+4} \ldots ob_1$ ob_{n+4}

`DROP#1-` #637F4h
Drops one object from the stack, then subtracts 1 from #x
#x ob $\rightarrow$ #x-1

`DROPONE` #62946h
Replaces object with #1
ob $\rightarrow$ #1

`DUP3PICK#+` #63704h
Duplicates #y, copies #x, then adds
#x #y $\rightarrow$ #x #y #x+y

`DUP#1+` #628EBh
Duplicates #x, then adds 1
#x $\rightarrow$ #x #x+1

`DUP#1+PICK` #6119Eh
Duplicates #n, adds 1, then does PICK
$ob_{n+1} \ldots ob_1$ #n $\rightarrow$ $ob_{n+1} \ldots ob_1$ #n ob_{n+1}

`DUP#1-`			#6292Fh
Duplicates #x, then subtracts 1			
#x	→	#x #x-1	
`DUP#2+`			#626F7h
Duplicates #x, then adds 2			
#x	→	#x #x+2	
`DUPTWO`			#63AD8h
Duplicates ob, then returns #2			
ob	→	ob ob #2	
`DUPZERO`			#63A88h
Duplicates ob, then returns 0			
ob	→	ob ob #0	
`OVER#+`			#6372Ch
Copies #x, then adds to #y			
#x #y	→	#x #x+y	
`OVER#-`			#6377Ch
Copies #x, then subtracts from #y			
#x #y	→	#x #y-x	
`OVER#2+UNROL`			#63105h
Copies #n, adds 2, then does UNROLL			
ob_{n+2} ... ob_3 #n ob_1	→	$\mathrm{ob}_1\mathrm{ob}_{n+2}$ ... ob_3 #n	
`ROT#+`			#63718h
Moves #x to level 1, then adds to #y			
#x ob #y	→	ob #x+y	
`ROT#+SWAP`			#62DCCh
Moves #x to level 1, adds to #y, then swaps levels 1 and 2			
#x ob #y	→	#x+y ob	
`ROT#-`			#63768h
Moves #x to level 1, then subtracts from #y			
#x ob #y	→	ob #y-x	
`ROT#1+`			#637B8h
Moves #x to level 1, then adds 1			
#x ob_1 ob_2	→	$\mathrm{ob}_1\mathrm{ob}_2$ #x+1	
`SWAP#-`			#62794h
Swaps #x and #y, then subtracts #x from #y			
#x #y	→	#y-x	
`SWAP#1+`			#62904h
Moves #x to level 1, then adds 1			
#x ob	→	ob #x+1	
`SWAP#1+SWAP`			#51843h
Adds 1 to #x			
#x ob	→	#x+1 ob	
`SWAP#1-`			#637E0h
Swaps #x to level 1, then subtracts 1 from #x			
#x ob	→	ob #x-1	
`SWAP#1-SWAP`			#51857h
Subtracts 1 from #x in level 2			
#x ob	→	#x-1 ob	
`SWAPOVER#-`			#637A4h
Returns #y and #x-y			
#x #y	→	#y #x-y	
`ZEROOVER`			#63079h
Returns #0, then does OVER			
ob	→	ob #0 ob	

`ZEROSWAP`		#62E3Ah
Returns #0, then does SWAP		
ob	→	#0 ob

3.2 Flags

In User-RPL programs, the result of comparisons (like `>`) are real numbers with the value 0 or 1. In System-RPL programs test results are generally the built-in objects `TRUE` and `FALSE`. These flags are used for many purposes, most frequently branching decisions. When executed, these flags just put themselves on the stack:

`FALSE`		#03AC0h
The system object FALSE		
	→	FALSE
`TRUE`		#03A81h
The system object TRUE		
	→	TRUE

The objects `DROPTRUE` and `DROPFALSE` drop an object and place a flag on the stack:

`DROPFALSE`		#6210Ch
Replaces an object with FALSE		
ob	→	FALSE
`DROPTRUE`		#62103h
Replaces an object with TRUE		
ob	→	TRUE

Other objects are available that put two flags on the stack:

`FALSETRUE`		#6350Bh
Puts FALSE and TRUE on the stack		
	→	FALSE TRUE
`FalseFalse`		#2F934h
Puts two FALSE flags on the stack		
	→	FALSE FALSE
`TrueFalse`		#634F7h
Puts TRUE and FALSE on the stack		
	→	TRUE FALSE
`TrueTrue`		#0BBEDh
Puts two TRUE flags on the stack		
	→	TRUE TRUE

3.2.1 Flag Conversions

When either of these flags are displayed in the HP 48 stack display, you just see `External`. User-RPL tests return the real numbers 1 or 0 for TRUE or FALSE. The object `COERCEFLAG` is useful for converting flags to real numbers if your System-RPL program needs to return a true/false result when ending. `COERCEFLAG` returns 1 for `TRUE` or 0 for `FALSE`, then exits the current secondary.

`COERCEFLAG`		#5380Eh
Converts a system flag into a real number and exits the current secondary		
TRUE	→	%1
FALSE	→	%0

To convert a real number into a flag, use the object `%0<>`:

`%0<>`	#2A7CFh
Returns TRUE if a real number is non-zero	
% → FLAG	

The object `%0<>` is one member of a large family of test objects which are discussed in greater detail in *Tests* on the current page.

Example: This program fragment shows the use of `COERCEFLAG` in a program that needs to return a true/false result to the user at exit:

```
::              Start of program
   ...          Establish TRUE or FALSE flag on stack
   COERCEFLAG   Convert flag to 0 or 1
;               End of program
```

Example: This program fragment shows the use of `ITE` (if...then...else, described later) to return a true/false result to the user before going on to other tasks. `AtUserStack` marks the result as being "owned by the user", so that the result won't be discarded if an error occurs later on.

```
::
   ...           Establish TRUE or FALSE flag on stack
   ITE %1 %0     Use ITE to put the corresponding real number on the stack
   AtUserStack   Mark the result as being owned by the user
   ...           The program continues
;
```

Any time a System-RPL program returns a result to the user, the result should be marked so that it is preserved for the user in case of low memory or other errors. The use of `COERCEFLAG` is often one of these cases. The object `AtUserStack` is sometimes used for this purpose, and is discussed in *Argument Validation* on page 52.

3.2.2 Flag Utilities

The following objects are available for manipulating flags:

`AND`	#03B46h
Logical AND	
$FLAG_1 FLAG_2 \rightarrow FLAG_3$	
`NOT`	#03AF2h
Logical NOT	
$FLAG_1 \rightarrow FLAG_2$	
`ORNOT`	#635B0h
Logical OR followed by logical NOT	
$FLAG_1 FLAG_2 \rightarrow FLAG_3$	
`NOTAND`	#62C55h
Logical NOT, followed by logical AND	
$FLAG_1 FLAG_2 \rightarrow FLAG_3$	
`ROTAND`	#62C91h
Performs ROT, followed by logical AND	
$FLAG_1 ob\ FLAG_2 \rightarrow ob\ FLAG_3$	
`XOR`	#03ADAh
Logical XOR	
$FLAG_1 FLAG_2 \rightarrow FLAG_3$	

3.3 Tests

The internal flags TRUE and FALSE appear most frequently as the result of a test on one or more objects. The following objects test object equality, bints, real numbers, extended real numbers, and complex numbers. There are also tests for object types, listed under *Object Type Tests* on page 57.

3.3.1 Object Equality

There are two types of object equality tests:

- The `EQ` family tests to see if two objects are the same object — their physical addresses are identical.
- The `EQUAL` family test to see if two objects are equal — even if their physical addresses are not the same. This is the internal counterpart to the User-RPL command `SAME`.

Word	Description	Stack	Address
`EQ`	Returns TRUE if objects have the same physical address	$ob_2 ob_1 \rightarrow$ FLAG	#03B2Eh
`EQUAL`	Returns TRUE if objects are equal (like User-RPL `SAME`)	$ob_2 ob_1 \rightarrow$ FLAG	#03B97h
`2DUPEQ`	Returns TRUE if objects have the same physical address	$ob_2 ob_1 \rightarrow ob_2 ob_1$ FLAG	#635D8h
`EQOR`	Does EQ test, then ORs the result with FLAG	$FLAG_1 ob_2\ ob_1 \rightarrow FLAG_2$	#63605h
`EQOVER`	Does EQ test, then OVER	$ob_3 ob_2\ ob_1 \rightarrow ob_3$FLAG ob_3	#6303Dh
`EQUALNOT`	Performs EQUAL, followed by logical NOT	$ob_2 ob_1 \rightarrow$ FLAG	#635C4h
`EQUALOR`	Does EQUAL test, then logical OR	$FLAG_1 ob_2\ ob_1 \rightarrow FLAG_2$	#63619h

3.3.2 Binary Integer Tests

The following objects test the value of internal binary integers:

Word	Description	Stack	Address
`#=`	Equal	#x #y → FLAG	#03D19h
`#<>`	Not equal	#x #y → FLAG	#03D4Eh
`#>`	Greater than	#x #y → FLAG	#03D83h
`#<`	Less than	#x #y → FLAG	#03CE4h
`2DUP#<`	Duplicates #x and #y, then does less-than test	#x #y → #x #y FLAG	#6289Bh
`2DUP#=`	Duplicates #x and #y, then does equal test	#x #y → #x #y FLAG	#628B5h

`2DUP#>`	#628D1h
Duplicates #x and #y, then does greater-than test	
#x #y → #x #y FLAG	

`#0=`	#03CA6h
Returns TRUE if bint = <0>	
# → FLAG	

`#0<>`	#03CC7h
Returns TRUE if bint ≠ <0>	
# → FLAG	

`#1=`	#622A7h
Returns TRUE if bint = <1>	
# → FLAG	

`#1<>`	#622B6h
Returns TRUE if bint ≠ <1>	
# → FLAG	

`#2=`	#6229Ah
Returns TRUE if bint = <2>	
# → FLAG	

`#2<>`	#636C8h
Returns TRUE if bint ≠ <2>	
# → FLAG	

`#3=`	#62289h
Returns TRUE if bint = <3>	
# → FLAG	

`#5=`	#636B4h
Returns TRUE if bint = <5>	
# → FLAG	

`DUP#0<>`	#622D4h
Duplicates #, then returns TRUE if bint ≠ <0>	
# → # FLAG	

`DUP#0=`	#62266h
Duplicates #, then returns TRUE if bint = <0>	
# → # FLAG	

`DUP#1=`	#622C5h
Duplicates #, then returns TRUE if bint = <1>	
# → # FLAG	

`DUP#7<`	#63687h
Duplicates #, then returns TRUE if bint < <7>	
# → # FLAG	

`OVER#0=`	#622C5h
Returns TRUE if bint = <0>	
# ob → # ob FLAG	

3.3.3 Real Number Tests

The following objects compare the values of two real numbers:

`%<`	#2A871h
Less than	
$\%_2\ \%_1$ → FLAG	

`%<=`	#2A8B6h
Less than or equal	
$\%_2\ \%_1$ → FLAG	

`%<>`	#2A8CCh
Not equal	
$\%_2\ \%_1 \rightarrow$ FLAG	

`%=`	#2A8C1h
Equal	
$\%_2\ \%_1 \rightarrow$ FLAG	

`%>`	#2A88Ah
Greater than	
$\%_2\ \%_1 \rightarrow$ FLAG	

`%>=`	#2A8A0h
Greater than or equal	
$\%_2\ \%_1 \rightarrow$ FLAG	

`%MAXorder`	#62D81h
Orders two real numbers	
$\%_2\ \%_1 \rightarrow \%_{\text{largest}}\ \%_{\text{smallest}}$	

The following objects test the value of a single real number:

`%0<`	#2A738h
Less than zero	
$\% \rightarrow$ FLAG	

`%0<>`	#2A7CFh
Not equal to zero	
$\% \rightarrow$ FLAG	

`%0=`	#2A76Bh
Equal to zero	
$\% \rightarrow$ FLAG	

`%0>`	#2A799h
Greater than zero	
$\% \rightarrow$ FLAG	

`%0>=`	#2A7F7h
Greater than or equal to zero	
$\% \rightarrow$ FLAG	

`DUP%0=`	#63BAAh
Duplicates %, then does equal to zero test	
$\% \rightarrow \%$ FLAG	

3.3.4 Extended Real Number Tests

The following objects test the value of two extended real numbers:

`%%<`	#2A81Fh
Less than	
$\%\%_2\ \%\%_1 \rightarrow$ FLAG	

`%%<=`	#2A8ABh
Less than or equal	
$\%\%_2\ \%\%_1 \rightarrow$ FLAG	

`%%>`	#2A87Fh
Greater than	
$\%\%_2\ \%\%_1 \rightarrow$ FLAG	

`%%>=`	#2A895h
Greater than or equal	
$\%\%_2\ \%\%_1 \rightarrow$ FLAG	

The following objects test the value of an extended real number:

`%%0<=`		#2A80Bh
Less than or equal to zero		
%%	→	FLAG

`%%0<`		#2A727h
Less than zero		
%%	→	FLAG

`%%0<>`		#2A7BBh
Not equal to zero		
%%	→	FLAG

`%%0=`		#2A75Ah
Equal to zero		
%%	→	FLAG

`%%0>`		#2A788h
Greater than zero		
%%	→	FLAG

`%%0>=`		#2A7E3h
Greater than or equal to zero		
%%	→	FLAG

3.3.5 Complex Number Tests

The following two objects test the values of a complex number or an extended complex number:

`C%0=`		#51B43h
Equal to C%0		
C%	→	FLAG

`C%%0=`		#51B2Ah
Equal to C%%0		
C%%	→	FLAG

3.3.6 Advanced Topic: Missing Extended Real Test Objects

Notice that objects to perform the tests %%= and %%<> aren't included in the tests listed on the previous page. These objects don't exist because they weren't used in the HP 48 operating system, and thus were left out to save ROM space. These objects can be created with a tiny bit of assembly language. We include the assembly language examples EREQ and ERNEQ, which generate code objects to perform these tests.

EREQ.A

```
******************************************************************************
** Object:  EREQ
**
** Purpose:  Compare two extended real numbers, return TRUE if equal
**
** Entry: 2:  %%2 (Extended Real Number)
**        1:  %%1 (Extended Real Number)
**
** Exit:  1:  FLAG (TRUE if %%2=%%1)
**
******************************************************************************
     NIBASC     /HPHP48-A/
     CON(5)     =DOCODE
     REL(5)     end
     P=         2
     GOVLNG     (=%%<)+7
end
```

EREQ can be embedded in System-RPL source code as follows:

```
 ::
  ...
 CODE
        P=             2
        GOVLNG         (=%%<)+7
 ENDCODE
  ...
 ;
```

The object ERNEQ is similar to EREQ, except that the initial value for P is different:

ERNEQ.A

```
*******************************************************************************
** Object:  ERNEQ
**
** Purpose:  Compare two extended real numbers, return TRUE if not equal
**
** Entry: 2:  %%2 (Extended Real Number)
**        1:  %%1 (Extended Real Number)
**
** Exit:  1:  FLAG (TRUE if %%2<>%%1)
**
*******************************************************************************
     NIBASC     /HPHP48-A/
     CON(5)     =DOCODE
     REL(5)     end
     P=         13
     GOVLNG     (=%%<)+7
end
```

3.3.7 Unit Object Tests

The following objects test the values of unit objects, returning %1 for TRUE and %0 for FALSE.

UM#?	#0F598h
Returns %1 if unit objects are not equal	
unit_1 unit_2 $\rightarrow$ %	
UM<=?	#0F5D4h
Returns %1 if $\text{unit}_1 \leq \text{unit}_2$	
unit_1 unit_2 $\rightarrow$ %	
UM<?	#0F5ACh
Returns %1 if $\text{unit}_1 < \text{unit}_2$	
unit_1 unit_2 $\rightarrow$ %	
UM=?	#0F584h
Returns %1 if $\text{unit}_1 == \text{unit}_2$	
unit_1 unit_2 $\rightarrow$ %	
UM>=?	#0F5E8h
Returns %1 if $\text{unit}_1 \geq \text{unit}_2$	
unit_1 unit_2 $\rightarrow$ %	
UM>?	#0F5C0h
Returns %1 if $\text{unit}_1 > \text{unit}_2$	
unit_1 unit_2 $\rightarrow$ %	

Note that the System-RPL object U>NCQ may be used to help determine if two unit objects are dimensionally consistent — see *Dimensional Consistency* on page 103.

3.3.8 Character String Tests

The following objects test character strings:

`DUPNULL$?`	#63209h
Duplicates $, then returns TRUE if $ is empty	
$ → $ FLAG	
`NULL$?`	#0556Fh
Returns TRUE if $ is empty	
$ → FLAG	

3.3.9 Hex String Tests

The following objects compare two hex strings, returning `%1` for TRUE and `%0` for FALSE. These tests respect the user's wordsize setting.

`HXS==HXS`	#544D9h
Returns %1 if hex strings are equal	
$\mathrm{hxs}_1\ \mathrm{hxs}_2 \rightarrow \%$	
`HXS#HXS`	#544ECh
Returns %1 if hex strings are not equal	
$\mathrm{hxs}_1\ \mathrm{hxs}_2 \rightarrow \%$	
`HXS<HXS`	#54552h
Returns %1 if $\mathrm{hxs}_1 < \mathrm{hxs}_2$	
$\mathrm{hxs}_1\ \mathrm{hxs}_2 \rightarrow \%$	
`HXS<=HXS`	#5453Fh
Returns %1 if $\mathrm{hxs}_1 \leq \mathrm{hxs}_2$	
$\mathrm{hxs}_1\ \mathrm{hxs}_2 \rightarrow \%$	
`HXS>=HXS`	#5452Ch
Returns %1 if $\mathrm{hxs}_1 \geq \mathrm{hxs}_2$	
$\mathrm{hxs}_1\ \mathrm{hxs}_2 \rightarrow \%$	
`HXS>HXS`	#54500h
Returns %1 if $\mathrm{hxs}_1 > \mathrm{hxs}_2$	
$\mathrm{hxs}_1\ \mathrm{hxs}_2 \rightarrow \%$	

3.4 Program Flow Control

We have already stated that programming in System-RPL is much like User-RPL, but there are more options for managing program execution in System-RPL. Before going further, it is important to highlight one major difference between the two environments. In User-RPL, an embedded program is *treated as an object* (e.g., placed on the stack), and in System-RPL an embedded secondary is *executed*. To illustrate the difference, consider the following two programs:

User-RPL:

```
«
  1
  « 2 »
  « 3 »
  4
«
```

Stack after execution:

System-RPL:

```
::
  %1
  :: %2 ;
  :: %3 ;
  %4
;
```

Stack after execution:

In combination with test objects that return TRUE or FALSE flags, we can take advantage of System-RPL's threaded execution to a great extent. Three classes of control objects are available:

- Objects that exit a secondary based on the state of a flag
- Object that support IF – THEN or IF – THEN – ELSE functions
- Objects that exit a secondary based on the state of a flag and perform additional actions prior to resuming execution of the parent secondary

Each of these classes of objects will be described and illustrated below.

3.4.1 Early Exits From a Secondary

The objects `?SEMI` and `NOT?SEMI` provide for early exits from a secondary based on the state of a flag on the stack. The object `#0=?SEMI` combines the `#0=` test with `?SEMI`, making one efficient object.

`?SEMI`	#61A3Bh
Exits the current secondary if FLAG is TRUE	
FLAG $\rightarrow$	

`NOT?SEMI`	#61A2Ch
Exits the current secondary if FLAG is FALSE	
FLAG $\rightarrow$	

`#0=?SEMI`	#61A18h
Exits the current secondary if # is zero	
# $\rightarrow$	

Example: The following embedded secondary divides a number by two and adds one to the result if it isn't zero:

```
::
  ...
  ::               Begin embedded secondary
    DUP%0= ?SEMI   Exit if real number is zero
    %2 %/ %1 %+    Complete calculation
  ;                End of embedded secondary
  ...
;
```

3.4.2 IF – THEN – ELSE Structures

There are two classes of objects that may be used to control program execution based on a system flag:

- Postfix objects that take their arguments from the stack
- Prefix objects that execute or skip the next object in the secondary

Postfix Objects. The postfix objects `RPIT` and `RPITE` take their arguments from the stack:

`RPIT`			#070FDh
Executes ob if FLAG is TRUE, otherwise drops ob			
TRUE ob	$\rightarrow$	*Executes ob*	
FALSE ob	$\rightarrow$		

`RPITE`			#070C3h
Execute $\mathrm{ob}_{\mathrm{TRUE}}$ if FLAG is TRUE, otherwise executes $\mathrm{ob}_{\mathrm{FALSE}}$			
TRUE $\mathrm{ob}_{\mathrm{TRUE}}$ $\mathrm{ob}_{\mathrm{FALSE}}$	$\rightarrow$	*Executes* ob_{TRUE}	
FALSE $\mathrm{ob}_{\mathrm{TRUE}}$ $\mathrm{ob}_{\mathrm{FALSE}}$	$\rightarrow$	*Executes* ob_{FALSE}	

Example: The following secondary expects a real number on the stack and puts "Zero" on the stack if it's zero, or "Non-Zero" if the number is non-zero:

```
::
  %0= "Zero" "Non-Zero" RPITE
;
```

Prefix Objects. The prefix objects take a flag from the stack and execute or skip the next one or two objects in the secondary. Note that `NOT_IT` and `?SKIP` are two commonly used names for the same object.

`NOT_IT` or `?SKIP`		#0712Ah
If FLAG is TRUE, skips the next object in the secondary		
FLAG	$\rightarrow$	
`:: ... ?SKIP` *object* `... ;`		

`IT`		#619BCh
If FLAG is TRUE, executes the next object in secondary otherwise skips the next object		
FLAG	$\rightarrow$	
`:: ... IT` $object_{TRUE}$ `;`		

`ITE`		#61AD8h
If FLAG is TRUE, executes the next object in secondary and skips the following object, otherwise skips the next object and executes the following object		
FLAG	$\rightarrow$	
`:: ... ITE` $object_{TRUE}$ $object_{FALSE}$ `... ;`		

Examples: The following secondary expects a real number on the stack, divides it by two if it's non-zero, and duplicates the result.

```
::
  DUP%0= ?SKIP :: %2 %/ ; DUP
;
```

The following secondary expects a real number on the stack and puts "Zero" on the stack if it's zero, or "Non-Zero" if the number is non-zero, then duplicates the result:

```
::
  %0=
  ITE
    "Zero"
    "Non-Zero"
  DUP
;
```

Combination Objects. The following objects combine test and branch operations:

`#0=?SKIP`	#6333Ah
If # is zero, skips the next object in the secondary	
# $\rightarrow$	
`:: ... #0=?SKIP object ... ;`	

`#1=?SKIP`	#63353h
If # is one, skips the next object in the secondary, otherwise executes the next object	
# $\rightarrow$	
`:: ... #1=?SKIP object ... ;`	

`#>?SKIP`	#63399h
If #x > #y, skips the next object	
#x #y $\rightarrow$	
`:: ... #>?SKIP object ... ;`	

`?SKIPSWAP`	#62D9Fh
If FLAG is FALSE, swaps ob_1 and ob_2	
ob_2 ob_1 FALSE $\rightarrow$ ob_1 ob_2	
ob_2 ob_1 TRUE $\rightarrow$ ob_2 ob_1	
`:: ... ?SKIPSWAP ... ;`	

`#0=ITE`	#63E89h
If # is zero, executes the next object in the secondary and skips the following object, otherwise skips the next object and executes the following object	
# $\rightarrow$	
`:: ... #0=ITE` $\mathrm{object}_{\mathrm{TRUE}}$ $\mathrm{object}_{\mathrm{FALSE}}$ `... ;`	

`#<ITE`	#63E9Dh
If #x <#y, executes the next object in the secondary and skips the following object, otherwise skips the next object and executes the following object	
#x #y $\rightarrow$	
`:: ... #<ITE` $\mathrm{object}_{\mathrm{TRUE}}$ $\mathrm{object}_{\mathrm{FALSE}}$ `... ;`	

`#=ITE`	#62C2Dh
If #x = #y, executes the next object in the secondary and skips the following object, otherwise skips the next object and executes the following object	
#x #y $\rightarrow$	
`:: ... #=ITE` $\mathrm{object}_{\mathrm{TRUE}}$ $\mathrm{object}_{\mathrm{FALSE}}$ `... ;`	

`ANDITE`	#63E61h
If (FLAG1 AND FLAG2) is TRUE, executes the next object in the secondary and skips the following object, otherwise skips the next object and executes the following object	
FLAG1 FLAG2 $\rightarrow$	
`:: ... ANDITE` $\mathrm{object}_{\mathrm{TRUE}}$ $\mathrm{object}_{\mathrm{FALSE}}$ `... ;`	

`DUP#0=IT`	#63E48h
Duplicates #, then if # is zero executes the next object in the secondary	
# $\rightarrow$ #	
`:: ... DUP#0=IT object ... ;`	

`DUP#0=ITE`	#63EC5h
Duplicates #, then if # is zero executes the next object in the secondary and skips the following object, otherwise skips the next object and executes the following object	
# $\rightarrow$ #	
`:: ... DUP#0=ITE` $\mathrm{object}_{\mathrm{TRUE}}$ $\mathrm{object}_{\mathrm{FALSE}}$ `... ;`	

`EQIT`	#63E2Fh
If ob_1 has the same address as ob_2, executes the next object in the secondary	
$\mathrm{ob}_2\mathrm{ob}_1$ $\rightarrow$	
`:: ... EQIT object ... ;`	

`EQITE`	#63E75h
If ob_1 has the same address as ob_2, executes the next object in the secondary and skips the following object, otherwise skips the next object and executes the following object	
$ob_2 ob_1 \rightarrow$	
`:: ... EQITE` object$_{TRUE}$ object$_{FALSE}$ `... ;`	

`SysITE`	#63EEDh
If the system flag specified by # is set, executes the next object in the secondary and skips the following object, otherwise skips the next object and executes the following object. System flags are numbered from #1d to #64d, corresponding to flags –1 to –64 in User-RPL.	
#system-flag $\rightarrow$	
`:: ... SysITE` object$_{TRUE}$ object$_{FALSE}$ `... ;`	

`UserITE`	#63ED9h
If the user flag specified by # is set, executes the next object in the secondary and skips the following object, otherwise skips the next object and executes the following object. User flags are numbered from #d to #64d, corresponding to flags 1 to 64 in User-RPL.	
#user-flag $\rightarrow$	
`:: ... UserITE` object$_{TRUE}$ object$_{FALSE}$ `... ;`	

Example: The following program tests system flag 40 to see if the clock is being displayed. The string “Program Complete” is appended with the time of day if the clock is being displayed, otherwise the string is appended with a period.

```
TIMEDONE 78.5 Bytes Checksum #2E17h
( → $ )
::
OLASTOWDOB!              Clears saved command name (see Argument Validation on page 52)
CKONOLASTWD              Asserts no arguments
"Program complete"
FORTY SysITE             Test system flag 40
  ::                     Start of TRUE object
    " at "                 " at "
    TOD TOD>t$ &$          Appends a string representing the current time of day to " at "
  ;                      End of TRUE object
  "."                    FALSE object
  &$                     Appends time or period string
;
```

3.4.3 CASE Objects

The object case provides one of the most useful program flow control options in System-RPL. `case` takes a flag from the stack, usually the result of a test operation. If the flag is TRUE, one level of the return stack is dropped (effectively discarding the rest of the secondary), then the next object in the secondary is executed. If the flag is FALSE, the next object in the secondary is skipped and the rest of the secondary is executed.

`case`	#61993h
If FLAG is TRUE, executes object$_{TRUE}$ and skips the remainder of the secondary, otherwise skips object$_{TRUE}$ and executes the remainder of the secondary	
FLAG $\rightarrow$	
`:: ... case` object$_{TRUE}$ `... ;`	

Example: The following secondary expects a real number on the stack, converts it to a bint, and returns "Zero" if the bint is 0, "One" if the bint is one, "Two" if the bint is two, otherwise returns "Other". This example validates the input argument using objects described in *Argument Validation* on page 52.

```
CASE1 97 Bytes Checksum #636Eh
( % → $ )
::
  0LASTOWDOB! CK1NOLASTWD         Expect one argument
  CK&DISPATCH1 real               Insist on a real number
  ::
    COERCE                        Convert real number to a bint
    DUP#0= case :: DROP "Zero" ;  Return "Zero" if bint is zero
    DUP#1= case :: DROP "One" ;   Return "One" if bint is one
    #2= case "Two"                Return "Two" if bint is two
    "Other"                       Return "Other" for all other values
  ;
;
```

CASE Combination Objects. There are many objects that can help save code by combining test or other operations with case. There are two classes of combination objects involving case:

- Objects that execute the next object and discard the remainder of the secondary if the flag is TRUE or skip the next object in the secondary and execute the remainder of the secondary if the flag is FALSE
- Objects that exit the secondary with an included action if the flag is TRUE or execute the remainder of the secondary if the flag is FALSE.

A naming convention helps to differentiate between the different case objects. Generally, an object name ending with DROP (capital letters) suggests an object whose last action is to DROP an object from the stack. Objects with drop in the name (lowercase) suggest an object that drops an object in the true case before performing the next task. Compare `casedrop` with `caseDROP` to see how this works.

Before listing the stack diagrams for these objects, we illustrate the use of four of them with examples.

The object `casedrop` combines `case` with the action of `DROP` before the true-object is executed:

`casedrop`			#618F7h
If FLAG is TRUE, drops an object from the stack, executes $\mathrm{object}_{\mathrm{TRUE}}$, and skips the remainder of the secondary; otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary			
ob TRUE	→		
ob FALSE	→	ob	
`:: ... casedrop object`$_{\mathrm{TRUE}}$ `... ;`			

The object `DUP#0=csedrp` combines the actions of `DUP#0=` and `casedrop` into one object:

`DUP#0=csedrp`				#618A8h
Duplicates #, then if # is zero, drops # from the stack, executes $\mathrm{object}_{\mathrm{TRUE}}$, and skips the remainder of the secondary; otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary				
#	→		*(# = 0)*	
#	→	#	*(# $\neq$ 0)*	
`:: ... DUP#0=csedrp object`$_{\mathrm{TRUE}}$ `... ;`				

These combination objects allow you to rewrite the example CASE1 on the previous page saving 17.5 bytes:

```
CASE2 79.5 Bytes Checksum #BEF2h
( % → $ )
::
  0LASTOWDOB! CK1NOLASTWD        Expect one argument
  CK&DISPATCH1 real              Insist on a real number
  ::
    COERCE                       Convert real number to a bint
    DUP#0=csedrp "Zero"          Return "Zero" if bint is zero
    DUP#1= casedrop "One"        Return "One" if bint is one
    #2= case "Two"               Return "Two" if bint is two
    "Other"                      Return "Other" for all other values
  ;
;
```

The object `#=casedrop` combines the actions `OVER`, `#=`, and `casedrop` into a single object that's useful for executing different objects based on the value of a bint. This object is used frequently in key handlers, and probably should have been named `OVER#=casedrop`.

`#=casedrop`			#618D3h
If #x = #y, drops #x and #y from the stack, executes $\mathrm{object}_{\mathrm{TRUE}}$, and skips the remainder of the secondary, otherwise drops #y, skips $\mathrm{object}_{\mathrm{TRUE}}$, and executes the remainder of the secondary			
#x #y	→		*(#x = #y)*
#x #y	→	#x	*(#x ≠ #y)*
`:: ... #=casedrop object`$_{\mathrm{TRUE}}$ `... ;`			

The example CASE3 uses `#=casedrop` to produce another variant on our previous two examples:

```
CASE3 82 Bytes Checksum #89E0h
( % → $ )
::
  0LASTOWDOB! CK1NOLASTWD        Expect one argument
  CK&DISPATCH1 real              Insist on a real number
  ::
    COERCE                       Convert real number to a bint
    ZERO #=casedrop "Zero"       Return "Zero" if bint is zero
    ONE #=casedrop "One"         Return "One" if bint is one
    #2= case "Two"               Return "Two" if bint is two
    "Other"                      Return "Other" for all other values
  ;
;
```

The second class of `case` combination objects mentioned is objects that exit with a combined operation or execute the remainder of the secondary. An example of this is `caseDROP`.

`caseDROP`			#6194Bh
If FLAG is TRUE, drops an object from the stack and exits the secondary; otherwise executes the remainder of the secondary			
ob TRUE	→		
ob FALSE	→	ob	
`:: ... caseDROP ... ;`			

Example: This secondary expects a real number on the stack representing a user flag. If the number is in the range 1 to 4, the corresponding user flag is set, otherwise no action is taken.

CASE4 49.5 Bytes Checksum #DCA7h
(% →)

```
::
  0LASTOWDOB! CK1NOLASTWD        Expect one argument
  CK&DISPATCH1 real              Insist on a real number
  ::
    COERCE                       Convert real number to a bint
    DUP#0= caseDROP              Exit, dropping the hint, if the hint is zero
    DUP FOUR #> caseDROP         Exit, dropping the hint, if the bint is greater than four
    SetUserFlag                  Set the user flag
  ;
;
```

Here are the objects that combine case with other operations:

`#=casedrop` #618D3h

If #x = #y, drops #x and #y from the stack, executes $\mathrm{object}_{\mathrm{TRUE}}$, and skips the remainder of the secondary, otherwise drops #y, skips $\mathrm{object}_{\mathrm{TRUE}}$, and executes the remainder of the secondary

#x #y → *(#x = #y)*
#x #y → #x *(#x ≠ #y)*

`:: ... #=casedrop objectTRUE ... ;`

`%0=case` #5F127h

If % is equal to zero, executes $\mathrm{object}_{\mathrm{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary

% →

`:: ... %0=case objectTRUE ... ;`

`%1=case` #5F181h

If % is equal to one, executes $\mathrm{object}_{\mathrm{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary

% →

`:: ... %1=case objectTRUE ... ;`

`ANDNOTcase` #63DDFh

If FLAG_1 and FLAG_2 are not both TRUE, executes $\mathrm{object}_{\mathrm{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary

FLAG_2 FLAG_1 →

`:: ... ANDNOTcase objectTRUE ... ;`

`ANDcase` #63CEAh

If FLAG_1 and FLAG_2 are both TRUE, executes $\mathrm{object}_{\mathrm{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary

FLAG_2 FLAG_1 →

`:: ... ANDcase objectTRUE ... ;`

`DUP#0=case` #61891h

Duplicates #, then if # is zero executes $\mathrm{object}_{\mathrm{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary

→

`:: ... DUP#0=case objectTRUE ... ;`

`DUP#0=csedrp`	#618A8h
Duplicates #, then if # is zero, drops # from the stack, executes $\mathrm{object_{TRUE}}$, and skips the remainder of the secondary, otherwise skips $\mathrm{object_{TRUE}}$ and executes the remainder of the secondary $\# \rightarrow$ (# = 0) $\# \rightarrow \#$ ($\# \neq 0$) `:: ... DUP#0=csedrp object`$_{\mathtt{TRUE}}$` ... ;`	

`EQUALNOTcase`	#63DF3h
If $\mathrm{ob_1}$ is not equal to $\mathrm{ob_2}$, executes $\mathrm{object_{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object_{TRUE}}$ and executes the remainder of the secondary $\mathrm{ob_2}\ \mathrm{ob_1} \rightarrow$ `:: ... EQUALNOTcase object`$_{\mathtt{TRUE}}$` ... ;`	

`EQUALcase`	#63CFEh
If $\mathrm{ob_1}$ is equal to $\mathrm{ob_2}$, executes $\mathrm{object_{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object_{TRUE}}$ and executes the remainder of the secondary $\mathrm{ob_2}\ \mathrm{ob_1} \rightarrow$ `:: ... EQUALcase object`$_{\mathtt{TRUE}}$` ... ;`	

`EQUALcasedrp`	#63CA4h
If $\mathrm{ob_1}$ is equal to $\mathrm{ob_2}$, drops $\mathrm{ob_1}$ from the stack, executes $\mathrm{object_{TRUE}}$, and skips the remainder of the secondary, otherwise skips $\mathrm{object_{TRUE}}$ and executes the remainder of the secondary $\mathrm{ob_3}\ \mathrm{ob_2}\ \mathrm{ob_1} \rightarrow$ ($ob_1 = ob_2$) $\mathrm{ob_3}\ \mathrm{ob_2}\ \mathrm{ob_1} \rightarrow \mathrm{ob_1}$ ($ob_1 \neq ob_2$) `:: ... EQUALcasedrp object`$_{\mathtt{TRUE}}$` ... ;`	

`EQcase`	#61933h
If $\mathrm{ob_1}$ has the same address as $\mathrm{ob_2}$, executes $\mathrm{object_{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object_{TRUE}}$ and executes the remainder of the secondary $\mathrm{ob_2}\ \mathrm{ob_1} \rightarrow \mathrm{ob_2}$ `:: ... EQcase object`$_{\mathtt{TRUE}}$` ... ;`	

`NOTcase`	#619ADh
If FLAG is FALSE, executes $\mathrm{object_{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object_{TRUE}}$ and executes the remainder of the secondary FLAG $\rightarrow$ `:: ... NOTcase object`$_{\mathtt{TRUE}}$` ... ;`	

`NOTcasedrop`	#618E8h
If FLAG is FALSE, drops ob, executes $\mathrm{object_{TRUE}}$, and skips the remainder of the secondary, otherwise skips $\mathrm{object_{TRUE}}$ and executes the remainder of the secondary TRUE $\rightarrow$ ob FALSE $\rightarrow$ `:: ... NOTcasedrop object`$_{\mathtt{TRUE}}$` ... ;`	

`NOTcase2drop`	#619ADh
If FLAG is FALSE, drops $\mathrm{ob_1}$ and $\mathrm{ob_2}$, executes $\mathrm{object_{TRUE}}$, and skips the remainder of the secondary, otherwise skips $\mathrm{object_{TRUE}}$ and executes the remainder of the secondary TRUE $\rightarrow$ $\mathrm{ob_2}\ \mathrm{ob_1}$ FALSE $\rightarrow$ `:: ... NOTcase2drop object`$_{\mathtt{TRUE}}$` ... ;`	

`ORcase`	#629BCh
If either FLAG_1 or FLAG_2 are TRUE, executes $\mathrm{object}_{\mathrm{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary	
$\mathrm{FLAG}_2\ \mathrm{FLAG}_1 \rightarrow$	
`:: ... ORcase objectTRUE ... ;`	

`OVER#=case`	#6187Ch
Does OVER, then if #1 = #2, executes $\mathrm{object}_{\mathrm{TRUE}}$ and skips the remainder of the secondary, otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary	
#2 #1 → #2	
`:: ... OVER#=case objectTRUE ... ;`	

`casedrop`	#618F7h
If FLAG is TRUE, drops an object from the stack, executes $\mathrm{object}_{\mathrm{TRUE}}$, and skips the remainder of the secondary, otherwise skips $\mathrm{object}_{\mathrm{TRUE}}$ and executes the remainder of the secondary	
ob TRUE →	
ob FALSE → ob	
`:: ... casedrop objectTRUE ... ;`	

The following `case` combination objects execute an action before skipping the remainder of the current secondary if the flag argument or test result is true.

`DUP#0=csDROP`	#618A8h
Duplicates #, then if # = 0, drops # and skips the remainder of the secondary	
# →	*(# = 0)*
# → #	*(# ≠ 0)*
`:: ... DUP#0=csDROP ... ;`	

`NOTcase2DROP`	#61984h
If FLAG is FALSE, drops two objects from the stack and skips the remainder of the secondary	
$\mathrm{ob}_2\ \mathrm{ob}_1$ TRUE → $\mathrm{ob}_2\mathrm{ob}_1$	
$\mathrm{ob}_2\ \mathrm{ob}_1$ FALSE →	
`:: ... NOTcase2DROP ... ;`	

`NOTcaseFALSE`	#5FB49h
If FLAG is TRUE, executes the remainder of the secondary, otherwise puts FALSE on the stack and skips the remainder of the secondary	
TRUE →	
FALSE → FALSE	
`:: ... NOTcaseFALSE ... ;`	

`NOTcaseTRUE`	#638CBh
If FLAG is TRUE, executes the remainder of the secondary, otherwise puts TRUE on the stack and skips the remainder of the secondary	
TRUE →	
FALSE → TRUE	
`:: ... NOTcaseTRUE ... ;`	

`NcaseSIZEERR`	#63B19h
If FLAG is TRUE, executes the remainder of the secondary, otherwise issues the `Bad Argument Value` error	
FLAG →	
`:: ... NcaseSIZEERR ... ;`	

`NcaseTYPEERR`	#63B46h
If FLAG is TRUE, executes the remainder of the secondary, otherwise issues the `Bad Argument Type` error	
FLAG →	
`:: ... NcaseTYPEERR ... ;`	

`case2DROP`	#61984h
If FLAG is TRUE, drops two objects from the stack and skips the remainder of the secondary	
ob_2 ob_1 TRUE →	
ob_2 ob_1 FALSE → $ob_2 ob_1$	
`:: ... case2DROP ... ;`	

`caseDROP`	#6194Bh
If FLAG is TRUE, drops an object from the stack and skips the remainder of the secondary	
ob TRUE →	
ob FALSE → ob	
`:: ... caseDROP ... ;`	

`caseDoBadKey`	#63BEBh
If FLAG is TRUE, executes DoBadKey (issues invalid key beep) and skips the remainder of the secondary	
FLAG →	
`:: ... caseDoBadKey ... ;`	

`caseDrpBadKy`	#63BD2h
If FLAG is TRUE, drops an object from the stack, executes DoBadKey (issues invalid key beep), and skips the remainder of the secondary	
ob TRUE →	
ob FALSE → ob	
`:: ... caseDrpBadKy ... ;`	

`caseERRJMP`	#63169h
If FLAG is TRUE, skips the remainder of the secondary and does ERRJMP	
FLAG →	
`:: ... caseERRJMP ... ;`	

`caseFALSE`	#6359Ch
If FLAG is TRUE, puts FALSE on the stack and skips the remainder of the secondary	
FALSE →	
TRUE → FALSE	
`:: ... caseFALSE ... ;`	

`caseSIZEERR`	#63B05h
If FLAG is FALSE, executes the remainder of the secondary, otherwise issues the `Bad Argument Value` error	
FLAG →	
`:: ... caseSIZEERR ... ;`	

`caseTRUE`	#634E3h
If FLAG is TRUE, puts TRUE on the stack and skips the remainder of the secondary	
FALSE →	
TRUE → TRUE	
`:: ... caseTRUE ... ;`	

`casedrpfls`	#6356Ah
If FLAG is TRUE, drops ob, puts FALSE on the stack, and skips the remainder of the secondary	
FALSE →	
ob TRUE → FALSE	
`:: ... casedrpfls ... ;`	

`case2drpfls`	#63583h
If FLAG is TRUE, drops ob_1 and ob_2, puts FALSE on the stack, and skips the remainder of the secondary	
FALSE →	
ob_2 ob_1 TRUE → FALSE	
`:: ... case2drpfls ... ;`	

`casedrptru` #628B2h
If FLAG is TRUE, drops ob, puts TRUE on the stack, and skips the remainder of the secondary
ob FALSE → ob
ob TRUE → TRUE
`:: ... casedrptru ... ;`

3.5 Loop Structures

Program loops are useful for repetitive execution of a procedure. There are two general classes of loops:

- *Definite loops* execute a *loop-clause* at least once, and execute a predefined number of iterations.
- *Indefinite loops* execute a *loop-clause* repeatedly until a *test-clause* returns a true result. One form of an indefinite loop may not execute at all if an initial test fails.

3.5.1 Definite Loops

Definite loops are implemented with the object `DO` and one of its counterparts: `LOOP` or `+LOOP`. When `DO` is executed, a DoLoop environment is created which stores the index and stopping value, and the interpreter pointer is copied to the return stack. The index and stop values are internal binary integers. DoLoop environments can be nested indefinitely.

Basic DoLoop Objects. The objects `DO`, `LOOP`, and `+LOOP` are recognized by the compiler RPLCOMP, which checks to see that `DO` and `LOOP` objects are properly matched.

`DO` #073F7h
Begins DO loop
#finish #start →
`:: ... #finish #start DO loop-clause LOOP ... ;`
`:: ... #finish #start DO loop-clause #increment +LOOP ... ;`

`LOOP` #07334h
Increments index of topmost DoLoop environment, abandons DoLoop environment if the new index is $\geq$ the stopping value, otherwise executes loop clause again
→

`#NAME?` #073A5h
Increments index of topmost DoLoop environment by #increment, abandons DoLoop environment if the new index is $\geq$ the stopping value, otherwise executes loop clause again
#increment →

DoLoop Utilities. The objects `#1+_ONE_DO`, `DUP#0_DO`, and `ZERO_DO` combine several actions into one object. When a program that uses these objects is being compiled with RPLCOMP, the compiler directive `(DO)` must be included after the object to tell the compiler that a DoLoop is being started. This will prevent an error from being generated when the compiler encounters the matching LOOP object.

`#1+_ONE_DO` #073DBh
Equivalent to `ONE #+ ONE DO`
#finish →
`:: ... #finish #1+_ONE_DO (DO) loop-clause LOOP ... ;`

`DUP#0_DO` #6347Fh
Equivalent to `DUP ZERO DO`
#finish → #finish
`:: ... #finish DUP#0_DO (DO) loop-clause LOOP ... ;`

`ZERO_DO`	#073C3h
Equivalent to `ZERO DO`	
#finish →	
`:: ... ` *#finish* `ZERO_DO (DO)` *loop-clause* `LOOP ... ;`	

Example: The following source fragment illustrates the use of these objects with the (DO) compiler directive:

```
::
  ...
  ZERO_DO (DO)
  ...
  LOOP
  ...
;
```

Accessing DoLoop Indices. The index value for the topmost DoLoop environment can be recalled with `INDEX@` and can be modified by using `INDEXSTO`. The index value for the second DoLoop environment can be recalled with `JINDEX@` and can be modified by using `JINDEXSTO`.

`INDEX@`	#07221h
Recalls the index value from the topmost DoLoop environment	
→ #index	
`INDEXSTO`	#07270h
Stores a new value for the index in the topmost DoLoop environment	
#index →	
`JINDEX@`	#07258h
Recalls the index value from the second DoLoop environment	
→ #index	
`JINDEXSTO`	#072ADh
Stores a new value for the index in the second DoLoop environment	
#index →	

Examples: The first program places the internal binary integers 4, 5, 6, and 7 on the stack; the second program places the internal binary integers 10, 20, and 30 on the stack:

```
:: EIGHT FOUR DO INDEX@ LOOP ;

:: THIRTYONE TEN DO INDEX@ TEN +LOOP ;
```

Accessing DO Loop Stop Values. The stop value for the topmost DoLoop environment can be recalled with `ISTOP@` and can be modified by using `ISTOPSTO`. The stop value for the second DoLoop environment can be recalled with `JSTOP@` and can be modified by using `JSTOPSTO`.

`ISTOP@`	#07249h
Recalls the stop value from the topmost DoLoop environment	
→ #stop	
`ISTOPSTO`	#07295h
Stores a new stop value in the topmost DoLoop environment	
#stop →	
`ZEROISTOPSTO`	#6400Fh
Stores <0d> in the stop value in the topmost DoLoop environment	
→	
`JSTOP@`	#07264h
Recalls the stop value from the second DoLoop environment	
→ #stop	
`JSTOPSTO`	#072C2h
Stores a new stop value in the second DoLoop environment	
#stop →	

3.5.2 Indefinite Loops

There are three indefinite loop structures available:

- `BEGIN` ... `WHILE` ... `REPEAT` loops contain an explicit test-clause and loop-clause. The loop clause may never be executed if the test-clause returns FALSE. The loop clause is assumed to be a secondary object — the RPLCOMP compiler places `::` and `;` around the loop clause. See *Compiling WHILE Loops* on the next page.
- `BEGIN` ... `UNTIL` loops always execute at least once — the object `UNTIL` expects either a TRUE or FALSE flag.
- `BEGIN` ... `AGAIN` loops have no test — they execute until an error event occurs or an `RDROP` is executed to remove the address placed on the return stack by `BEGIN`.

`AGAIN`	#071ABh
Unconditionally repeats loop-clause $\rightarrow$ `:: ... BEGIN` *loop-clause* `AGAIN ... ;`	
`BEGIN`	#071A2h

Copies the interpreter pointer to the return stack, serving as a beginning object for three loop structures
$\rightarrow$
`:: ... BEGIN` *loop-clause* `AGAIN ... ;`
`:: ... BEGIN` *test-clause* `WHILE` *loop-clause* `REPEAT ... ;`
`:: ... BEGIN` *loop-clause* `UNTIL ... ;`	

`REPEAT`	#071E5h
Copies the first pointer on the return stack to the interpreter pointer, completing a WHILE loop $\rightarrow$ `:: ... BEGIN` *test-clause* `WHILE` *loop-clause* `REPEAT ... ;`	
`WHILE`	#071EEh
If flag is true, allows execution of loop clause, otherwise drops one pointer from the return stack and skips the interpreter pointer to the object after `REPEAT` FLAG $\rightarrow$ `:: ... BEGIN` *test-clause* `WHILE` *loop-clause* `REPEAT ... ;`	
`UNTIL`	#071C8h
If flag is true, drops the top pointer on the return stack to terminate the loop, otherwise copies the first pointer on the return stack to the interpreter pointer to execute the loop-clause again FLAG $\rightarrow$ `:: ... BEGIN` *loop-clause* `UNTIL ... ;`	

Example: The following program returns the number of random numbers generated before one with a value greater than or equal to .95 is generated. The object `%RAN` (address #2AFC2h) returns a random number n such that $0 \leq n \leq 1$.

```
NUMRAN 53.5 Bytes Checksum #95D1h
( % → $ )
::
  AtUserStack          Clears saved command name, no arguments
  ZERO                 Initial value of the counter
  BEGIN                Beginning of WHILE loop structure
    %RAN % .95 %<      Test-clause
  WHILE                Executes loop-clause if flag is TRUE
    #1+                Loop-clause: increments counter
  REPEAT               Continue loop at %RAN
  UNCOERCE             Convert counter to real number
;
```

Compiling WHILE Loops. The RPLCOMP compiler places secondary delimiters around the loop clause in a WHILE loop. For instance, the example NUMRAN.S from the previous page looks like this after being compiled:

```
::
  AtUserStack
  ZERO
  BEGIN
    %RAN % .95 %<
  WHILE
    ::                  Beginning of secondary
      #1+
    ;                   End of secondary
  REPEAT
  UNCOERCE
;
```

Since the secondary delimiters are added by the compiler, you can use objects like `?SEMI` or `case` to cause an early exit from the loop clause (see *Case Structures* on page 41).

3.6 Runstream Operators

The return stack is a stack of pointers to objects embedded in composite objects, usually secondaries, called the runstream. The objects described here are useful for placing objects on the data or return stack, or for building your own control structures. The most often-used is `'`, which places the next object in the current secondary on the data stack.

`'` #06E97h
Pushes the next object (or object pointer) in the program on the data stack
→ object
`:: ... ' object ... ;`

`COLA` #06FD1h
Evaluates the next object in the current secondary, discarding the remainder of the secondary
→
`:: ... COLA object discarded objects ;`

`IDUP` #0716Bh
Copies the interpreter pointer to the return stack
→

`>R` #06EEBh
Pops a composite object off the data stack and pushes it on the return stack
`:: ... ;` →

`'R` #06F9Fh
Pops an object (or object pointer) off the return stack and pushes it on the data stack
→ object

`ticR` #61B89h
Pops the next object in the composite object in the return stack and pushes it and TRUE on the data stack. If the object is `SEMI`, pops the return stack and pushes FALSE on the data stack.
→ object TRUE *Not SEMI*
→ FALSE *SEMI*

`R@` #07012h
Creates a secondary in temporary memory (TEMPOB) from the composite pointed to by the top return stack pointer, pops the return stack, and pushes a pointer to the secondary on the return stack
→ `:: ... ;`

`R>`	#0701Fh
Creates a secondary in temporary memory (TEMPOB) from the composite pointed to by the top return stack pointer and pushes a pointer to the secondary on the return stack	
→ `:: ... ;`	

`RDROP`	#06FB7h
Pops the return stack	
→	

`2RDROP`	#6114Eh
Pops two levels off the return stack	
→	

`3RDROP`	#61160h
Pops three levels off the return stack	
→	

`RDUP`	#14EA5h
Duplicates the top item on the return stack	
→	

`RSWAP`	#60EBDh
Swaps the top two items on the return stack	
→	

The example RSTR in *Control Structure Examples* on page 72 shows how some of these objects may be used.

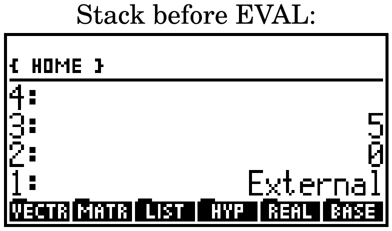
3.7 Argument Validation

Any program that is going accept input from the user should validate the number and type of arguments before proceeding. One of the reasons that you are probably interested in writing code in System-RPL is that you wish to avoid the argument checking that is inherent in every User-RPL command or function, yet it is still important to provide some protection at the very beginning.

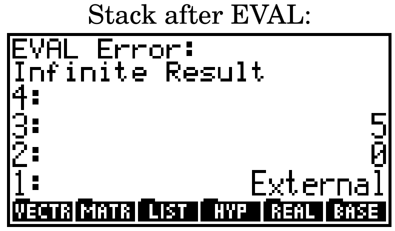
3.7.1 Attributing Errors

An integral part of the process of validating arguments is to make sure that errors are correctly attributed. This is often done in combination with type dispatching. To illustrate the problems associated with error attribution, consider the System-RPL program `:: %/ ;`. With the real numbers 5 and 0 in stack levels 3 and 2, and the object `:: %/ ;` in stack level 1, press [EVAL]. The divide operation generates an `Infinite Result` error:

Stack before EVAL:



Stack after EVAL:

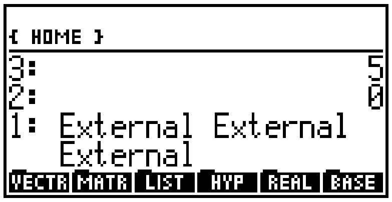


Notice that the error has been attributed to EVAL, which was the last object to claim responsibility for future errors. Further, the stack contents are not what you would expect. This can be solved by clearing out the saved command name (using `0LASTOWDOB!`) and checking for the proper number of arguments (using `CK2NOLASTWD`, described below).

`0LASTOWDOB!`	#1884Dh
Clears saved command name	
→	

The program now reads `:: OLASTOWDOB! CK2NOLASTWD %/ ;`. Now when you press [EVAL] a much more acceptable result appears:

Stack before EVAL:



Stack after EVAL:

If a program plans to accept no arguments, the object `AtUserStack` is a handy combination of `OLASTOWDOB!` and `CK0NOLASTWD` (described on the next page).

`AtUserStack`	#40BC9h
Requires no arguments, clears saved command name	
→	

3.7.2 Number of Arguments

The process for checking the number of arguments is slightly different for program objects that are being designed as stand alone applications vs. program objects that are included in a library application. The concept is the same in each case, however. (Library applications are discussed in the HP document MAKEROM.DOC and illustrated in GEOLIB example provided by HP. These are provided on the disk.) The structural outlines are:

System-RPL Programs	Library Commands

`::`
`OLASTOWDOB! CK`*n*`NOLASTWD`
`...`
`;`
	`::`
	`CK`*n*
	`...`
	`;`

where *n* refers to the number of arguments that are expected. The objects available for this task are:

System-RPL Program	Library Command	Number of Arguments
CK0NOLASTWD	CK0	No arguments required
CK1NOLASTWD	CK1	One argument required
CK2NOLASTWD	CK2	Two arguments required
CK3NOLASTWD	CK3	Three arguments required
CK4NOLASTWD	CK4	Four arguments required
CK5NOLASTWD	CK5	Five arguments required
CKNNOLASTWD	CKN	N arguments required

For instance, a System-RPL program that requires three objects on the stack should be structured as follows:

```
::
  OLASTOWDOB! CK3NOLASTWD
  ...
;
```

The objects `CKNNOLASTWD` and `CKN` are available for programs that take the number of arguments off the stack. Both objects convert the real number on the stack to an internal binary integer, then verify that the specified number of arguments are on the stack.

An example of this type of object is the User-RPL command `PICK`, in which a user-supplied real number specifies the stack level to copy. The code for the User-RPL `PICK` is `:: CKN PICK ;`, where the `PICK` is the internal System-RPL `PICK`.

Remember that in the case of library commands the `CK`*n* objects will attribute errors to the command name. System-RPL programs that are not parts of libraries or that need to ensure that their errors are not attributed to another command need to clear the saved command name. The objects `CK`*n*`NOLASTWD` do not modify the saved command name, so `OLASTOWDOB!` is needed to ensure that the saved command name will be cleared. This prevents an error generated in a program from being attributed to the last command that generated an error.

3.7.3 Type Dispatching

The HP 48's multiple polymorphic personality is attributable in part to the ability of each built-in command or function to interpret the types of arguments supplied and take meaningful action based on those types. The + function is one of the most dramatic examples, operating on over 20 different combinations of types of arguments.

The objects CK&DISPATCH0 and CK&DISPATCH1 perform a "check and dispatch" operation — choosing an object to be executed based on the types of stack arguments. The basic structure of a word using CK&DISPATCH*n* is:

```
::
  #type1 action1
  #type2 action2
  ...
  #typen actionn
;
```

where $\#\mathtt{type}_n$ is an internal binary integer encoding the desired object types, and $\mathtt{action}_n$ is the corresponding action to be taken when the arguments match the specified types. (Internal binary integers were discussed in greater detail in *Internal Binary Integers* on page 23.)

It is vital that the table of types and actions be terminated with ; . For System-RPL programs the basic structure for a program that has different actions based on argument types looks like this:

```
::
  0LASTOWDOB! CKnNOLASTWD
  CK&DISPATCHn
  #type1 action1
  ...
  #typen actionn
;
```

Since the table of actions must be terminated by ;, type dispatching operations embedded in larger programs should be set off in their own secondary. For example:

```
::
  ...
  ::
    CK&DISPATCH1
    # 00051 ::  Process list and real number  ;
    # 00041 ::  Process array and real number  ;
  ;
  ...
;
```

The example program GRID in *Graphics Examples* on page 129 illustrates the use of 0LASTOWDOB!, CK3NOLASTWD, and CK&DISPATCH1.

CK&DISPATCH0 vs. CK&DISPATCH1. In general, the HP 48 treats tags as auxiliary to the main purpose of any object, consequently CK&DISPATCH1 is used most frequently because it makes a second pass through the type-action table after recursively stripping any tags from the required objects. If it is important to type dispatch off tagged objects, then CK&DISPATCH0 should be used, which does not contain the second pass.

Type Dispatching in Library Applications. In the case of library commands, replacing each action with a pointer to an action will speed up the dispatch process because the time required to skip each action is reduced to the time required to skip a single pointer. For instance, the two examples below will do the same thing, but the example on the right will be slightly faster:

```
NULLNAME EX1
::
  CK2
  CK&DISPATCH1
  real :: ... ;
  cmp :: ... ;
  list :: ... ;
;
```

```
NULLNAME EX1
::
  CK2
  CK&DISPATCH1
  real EXSUB1
  cmp EXSUB2
  list EXSUB3
;

NULLNAME EXSUB1 :: ... ;
NULLNAME EXSUB2 :: ... ;
NULLNAME EXSUB3 :: ... ;
```

For library commands requiring at least one argument, the `CK`*n* and `CK&DISPATCH1` objects can be replaced with objects that combine their functionality:

Object	**Replaces**
CK1&Dispatch	CK1 CK&DISPATCH1
CK2&Dispatch	CK2 CK&DISPATCH1
CK3&Dispatch	CK3 CK&DISPATCH1
CK4&Dispatch	CK4 CK&DISPATCH1
CK5&Dispatch	CK5 CK&DISPATCH1

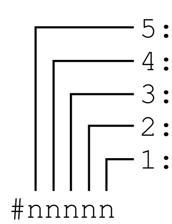
Using these objects, the examples above would look now like this:

```
NULLNAME EX1
::
  CK2&Dispatch
  real :: ... ;
  cmp :: ... ;
  list :: ... ;
;
```

```
NULLNAME EX1
::
  CK2&Dispatch
  real EXSUB1
  cmp EXSUB2
  list EXSUB3
;

NULLNAME EXSUB1 :: ... ;
NULLNAME EXSUB2 :: ... ;
NULLNAME EXSUB3 :: ... ;
```

Encoding Argument Types. The internal binary integer corresponding to each action can encode up to five object types. Viewed as five hex digits, the stack levels are specified as follows:



Each hex digit represents an argument type, as listed in the table on the next page. Notice that leading zeros mean that objects in their corresponding stack levels will be ignored. For instance, the internal binary integer `# 00051` specifies a list in level two and a real number in level one.

Some built-in binary integers can be used to encode individual objects or combinations of objects. In cases where a program is type-dispatching off of one argument, the built-in bints listed in the second column of the table may be used. For example, a program that takes different actions when the argument is a list or string might have the following structure:

```
::
  0LASTOWDOB! CK1NOLASTWD CK&DISPATCH1
  list :: ... ;
  str :: ... ;
;
```

Half of the objects that may be encoded require two digits. A program that requires an extended real in level two and an extended complex number in level one might have the following structure:

```
::
  0LASTOWDOB! CK2NOLASTWD
  CK&DISPATCH1
  # 03F4F :: ... ;
;
```

Encoding Digits	Built-in Bint	Object Type	User TYPE Number
0	`any`	Any Object	
1	`real`	Real Number	0
2	`cmp`	Complex Number	1
3	`str`	Character String	2
4	`arry`	Array	3,4
5	`list`	List	5
6	`idnt`	Global Name	6
7	`lam`	Local Name	7
8	`seco`	Secondary	8
9	`symb`	Symbolic	9
A	`sym`	Symbolic Class	6,7,9
B	`hxs`	Hex String	10
C	`grob`	Graphics Object	11
D	`TAGGED`	Tagged Object	12
E	`unitob`	Unit Object	13
0F		ROM Pointer	14
1F		Binary Integer	20
2F		Directory	15
3F		Extended Real	21
4F		Extended Complex	22
5F		Linked Array	23
6F	`char`	Character	24
7F		Code Object	25
8F		Library	16
9F		Backup	17
AF		Library Data	26
BF		External object1	27
CF		External object2	28
DF		External object3	29
EF		External object4	30

When possible, it is best to save code by using a built-in internal binary integer (2.5 bytes) instead of compiling a new one (5 bytes). The following built-in internal binary integers are used for type dispatching:

Name	Value	Name	Value
`2EXT`	`#000EEh`	`EXTREAL`	`#000E1h`
`2GROB`	`#000CCh`	`EXTSYM`	`#000EAh`
`2LIST`	`#00055h`	`REALEXT`	`#0001Eh`
`2REAL`	`#00011h`	`REALOB`	`#00010h`
`3REAL`	`#00111h`	`REALOBOB`	`#00100h`
`IDREAL`	`#00061h`	`REALREAL`	`#00011h`
`LISTCMP`	`#00052h`	`REALSYM`	`#0001Ah`
`LISTLAM`	`#00057h`	`ROMPANY`	`#000F0h`
`LISTREAL`	`#00051h`	`SYMBUNIT`	`#0009Eh`
`SYMREAL`	`#000A1h`	`SYMEXT`	`#000AEh`
`SYMSYM`	`#000AAh`	`SYMID`	`#000A6h`
`TAGGEDANY`	`#000D0h`	`SYMLAM`	`#000A7h`
`EXTOBOB`	`#00E00h`	`SYMOB`	`#000A0h`

3.7.4 Object Type Tests

There may be times when an initial test is not sufficient — a list must be in level one, but the contents of the list are also important. Two sets of objects are provided for System-RPL which are useful for testing the type of an object. These objects return the internal flags TRUE or FALSE (described in detail in *Tests* on page 31). The stack diagrams below illustrate the operation of the object tests:

`TYPEREAL?`		
Returns TRUE if object is a real number		
Object	→	FLAG

`DUPTYPEREAL?`		
Returns object and TRUE if object is a real number		
Object	→	Object FLAG

The objects in the first column test the type, returning a flag. The objects in the fourth column duplicate the object before testing the type.

Object type	Test Object	Address	Dup-and-Test Object	Address
Array	`TYPEARRY?`	#62198h	`DUPTYPEARRY?`	#62193h
Internal binary integer	`TYPEBINT?`	#6212Fh	`DUPTYPEBINT?`	#6212Ah
Complex array	`TYPECARRY?`	#62256h		
Character	`TYPECHAR?`	#62025h	`DUPTYPECHAR?`	#62020h
Complex number	`TYPECMP?`	#62183h	`DUPTYPECMP?`	#6217Eh
Program	`TYPECOL?`	#621ECh	`DUPTYPECOL?`	#621E7h
String	`TYPECSTR?`	#62159h	`DUPTYPECSTR?`	#62154h
Unit	`TYPEEXT?`	#6204Fh	`DUPTYPEEXT?`	#6204Ah
Graphics object	`TYPEGROB?`	#62201h	`DUPTYPEGROB?`	#621FCh
Hex string	`TYPEHSTR?`	#62144h	`DUPTYPEHSTR?`	#6213Fh
Identifier (global name)	`TYPEIDNT?`	#6203Ah	`DUPTYPEIDNT?`	#62035h
Temp. identifier (local name)	`TYPELAM?`	#6211Ah	`DUPTYPELAM?`	#62115h
List	`TYPELIST?`	#62216h	`DUPTYPELIST?`	#62211h
Real array	`TYPERARRY?`	#6223Bh		
Real number	`TYPEREAL?`	#6216Eh	`DUPTYPEREAL?`	#62169h
ROM pointer (XLIB name)	`TYPEROMP?`	#621ADh	`DUPTYPEROMP?`	#621A8h
Directory	`TYPERRP?`	#621C2h	`DUPTYPERRP?`	#621BDh
Symbolic	`TYPESYMB?`	#621D7h	`DUPTYPESYMB?`	#621D2h
Tagged	`TYPETAGGED?`	#6222Bh	`DUPTYPETAG?`	#62226h

Note: The objects `TYPECARRY?` and `TYPERARRY?` assume an array object is on the stack, and expect to find a prologue 10 nibbles into the object being tested.)

These tests can be helpful when the filtering provided by the check-and-dispatch mechanism does not provide a sufficient level of detail. For example, suppose a System-RPL program wants to ensure that it is processing a real number in level 2 and an array of real numbers in level one. The program shell might look like this:

```
::
  CK2NOLASTWD 0LASTOWDOB!
  CK&DISPATCH1
  # 00014
  ::
    DUP TYPERARRY? NcaseSIZEERR
  ...
  ;
;
```

This program would issue a `Bad Argument Value` error if the array was not an array of real numbers. The error is issued by the object `NcaseSIZEERR` if the flag on the stack is FALSE. Notice that the type checks for real and complex arrays don't have corresponding objects which first duplicate the object in question, so in this example the `DUP` had to be included.

3.8 Temporary Variables

Programs written in System-RPL have access to a much more flexible temporary (local) variable system than programs written in User-RPL. Temporary variables are stored in memory structures called "temporary environments". Like local variables in User-RPL, temporary variables can be very useful for cleaning up programs that otherwise would manage everything on the stack with great difficulty. In User-RPL, nested local variable environments are permitted, and the same goes for System-RPL. In System-RPL the creation of a temporary variable environment can happen at any time — it is not restricted to the beginning of a secondary. Temporary environments are stacked — they are abandoned in the reverse chronological order of their creation.

Remember:

- Temporary variables reside in temporary memory. When system garbage collection occurs, temporary memory is scanned and pointers to objects in temporary memory residing on the stack or in temporary variables are updated.
- When a temporary variable name is executed, the contents of the variable are recalled to the stack, but not executed.
- Storing to a temporary variable is typically quite fast, because temporary environments are typically small, and the system avoids the overhead of moving all the data in global variables.

In System-RPL, the object `BIND` does the job of → in User-RPL, and the object `ABND` does the job of » (actually named `x>>ABND` — you'll see this if you decompile a User-RPL program using a tool like Jazz). `BIND` expects the objects to be stored in temporary variables to be on the stack along with a list of temporary variable names in level one.

The object `DOBIND` does the work for `BIND` — the temporary variable names and their count are expected on the stack.

The RPL compiler creates a temporary variable name with the compiler directive `LAM`. For instance, to compile the temporary variable name "Fred", the compiler source should read `LAM Fred`. To save space, System-RPL also provides for null-named temporary variables (see *Using Null-Named Temporary Variables* on page 60). Space is saved because no name is stored and the temporary variables are referenced by number. The object `NULLLAM` may be used instead of a temporary variable name.

`BIND`	#074D0h
Creates a temporary environment	
$\mathrm{ob}_n \ldots \mathrm{ob}_2\ \mathrm{ob}_1$ { LAM $name_n$ ... LAM $name_2$ LAM $name_1$ } →	
$\mathrm{ob}_n \ldots \mathrm{ob}_2\ \mathrm{ob}_1$ { $\mathrm{NULLLAM}_n \ldots \mathrm{NULLLAM}_2\ \mathrm{NULLLAM}_1$ } →	

`DOBIND`	#074E4h
Creates a temporary environment	
$\mathrm{ob_n}$... $\mathrm{ob_2}$ $\mathrm{ob_1}$ **LAM** $name_n$ ... **LAM** $name_2$ **LAM** $name_1$ **#n** $\rightarrow$	
$\mathrm{ob_n}$... $\mathrm{ob_2}$ $\mathrm{ob_1}$ $\mathrm{NULLLAM_n}$ ... $\mathrm{NULLLAM_2}$ $\mathrm{NULLLAM_1}$ **#n** $\rightarrow$	

`ABND`	#07497h
Discards the topmost temporary environment	
$\rightarrow$	

When temporary variables are named, the process of storing to and recalling from temporary variables is the same as for User-RPL:

```
:: ... LAM Fred ... ;           Recalls the contents of temporary variable Fred
:: ... ' LAM Fred STO ... ;     Stores an object into temporary variable Fred
```

`STO`	#07D27h
Stores an object in a temporary variable	
object name $\rightarrow$	

There is no compiler requirement that there be a firm one-to-one matching between `BIND`s and `ABND`s. A secondary that has multiple exit points may need to have more than one `ABND` to ensure that temporary environments are discarded properly. The program QRT3 below illustrates this.

To compare the use of temporary variables in User-RPL and System-RPL, we'll begin by comparing two programs that do similar jobs — finding the roots of a quadratic equation $x = ax^2 + bx + c$. We'll use the quadratic formula:

$$\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

The stack diagram for these program examples will be:

a b c	$\rightarrow$	$\mathrm{root_1}$ $\mathrm{root_2}$

To keep things simple, the System-RPL examples will return the string `"Complex Roots"` if the quantity b^2-4ac is negative. (This is one of the attractive features of User-RPL: the polymorphic behavior of the operators lets you avoid writing extra code.)

We illustrate the use of temporary variables with four example programs. The first is written in User-RPL, the rest are written in System-RPL. The results are stored in temporary variables to illustrate the process, even though this is somewhat inefficient (the results could simply be left on the stack). Notice that this example uses compiled temporary variable `←a`, which will work only on HP 48G/GX calculators.

QRT1.RPL

```
«
  0 0 « ←a 2 * / »               Place zeros and subroutine on the stack
  → ←a b c root1 root2 Subr      Create temporary variables
  «
    b SQ ←a c * 4 * - √          Calculate SQRT(b² − 4ac)
    b NEG OVER + Subr EVAL       Calculate first root
    'root1' STO                  Store first root in local variable root1
    b NEG SWAP - Subr EVAL       Calculate second root
    'root2' STO                  Store second root in temporary variable root2
    root1 root2                  Return roots to the stack
  »                              Discards local variables
»
```

This is what QRT1.RPL looks like when expressed in System-RPL:

```
::
  x<<
    %0 %0 xSILENT' :: x<< LAM ←a %2 x* x/ x>> ;
    xRPN-> LAM ←a LAM b LAM c LAM root1 LAM root2 LAM Subr
    x<<
      LAM b xSQ LAM ←a LAM c x* %4 x* x- xSQRT
      LAM b xNEG xOVER x+ LAM Subr xEVAL
      x' LAM root1 xENDTIC xSTO
      LAM b xNEG xSWAP x- LAM Subr xEVAL
      x' LAM root2 xENDTIC xSTO
      LAM root1 LAM root2
    x>>ABND
  x>>
;
```

3.8.1 Using Named Temporary Variables

The first System-RPL example uses named temporary variables:

```
QRT1 250.5 Bytes Checksum #33EEh
( %a %b %c → %root1 %root2 )
```

```
::
  0LASTOWDOB! CK3NOLASTWD              Expect three arguments
  CK&DISPATCH1 3REAL                   Insist on three real numbers
  ::
    %0 %0                              Placeholder values for root1 and root2
    ' :: LAM a %2 %* %/ ;              Place subroutine on the stack
    {
      LAM a
      LAM b
      LAM c
      LAM root1
      LAM root2
      LAM Subr
    }                                  List of temporary variable names
    BIND                               Create temporary variable environment
    ::
      LAM b DUP %* LAM a LAM c %* %4 %* %-   Evaluate b^2 - 4ac
      DUP %0< casedrop "Complex Roots"  If <0, drop quantity, put string on stack, abandon temp
                                       env. and exit secondary
      %SQRT                            Evaluate SQRT(b^2 - 4ac)
      LAM b %CHS OVER %+ LAM Subr EVAL Calculate first root
      ' LAM root1 STO                  Store in root1
      LAM b %CHS SWAP %- LAM Subr EVAL Calculate second root
      ' LAM root2 STO                  Store in root2
      LAM root1                        Return first root to the user
      LAM root2                        Return second root to the user
    ;
    ABND                               Abandon temporary environment
  ;
;
```

3.8.2 Using Null-Named Temporary Variables

The second System-RPL example uses *null-named* temporary variables. When the object `NULLLAM` is used instead of a name, space is saved in the temporary environment. Access to null-named temporary variables is specified by the variable's number position in the temporary environment rather than by name. This kind of direct access is more efficient than searching through a series of names.

The objects PUTLAM and GETLAM are the fundamental tools used to store objects to and recall objects from temporary variables:

PUTLAM	#075E9h
Stores an object into numbered temporary variable	
object #variable →	
GETLAM	#075A5h
Recalls an object from a numbered temporary variable	
#variable → object	
NULLLAM	#34D30h
Null temporary variable name	
→ ''	

The use of PUTLAM and GETLAM can be streamlined by using objects which combine the bint specifying the temporary with the PUT or GET action. For instance, 2PUTLAM combines TWO PUTLAM into a single action that stores an object into the second temporary variable, and 4GETLAM combines FOUR GETLAM into a single object that recalls the object stored in the fourth temporary variable. These combined actions save code and are quite efficient.

PUTLAM Combinations		**GETLAM Combinations**	
Object	**Address**	**Object**	**Address**
1PUTLAM	#615E0h	1GETLAM	#613B6h
2PUTLAM	#615F0h	2GETLAM	#613E7h
3PUTLAM	#61600h	3GETLAM	#6140Eh
4PUTLAM	#61635h	4GETLAM	#61438h
5PUTLAM	#61625h	5GETLAM	#6145Ch
6PUTLAM	#61635h	6GETLAM	#6146Ch
7PUTLAM	#61645h	7GETLAM	#6147Ch
8PUTLAM	#61655h	8GETLAM	#6148Ch
9PUTLAM	#61665h	9GETLAM	#6149Ch
10PUTLAM	#61675h	10GETLAM	#614ACh
11PUTLAM	#61685h	11GETLAM	#614BCh
12PUTLAM	#61695h	12GETLAM	#614CCh
13PUTLAM	#616A5h	13GETLAM	#614DCh
14PUTLAM	#616B5h	14GETLAM	#614ECh
15PUTLAM	#616C5h	15GETLAM	#614FCh
16PUTLAM	#616D5h	16GETLAM	#6150Ch
17PUTLAM	#616E5h	17GETLAM	#6151Ch
18PUTLAM	#616F5h	18GETLAM	#6152Ch
19PUTLAM	#61705h	19GETLAM	#6153Ch
20PUTLAM	#61715h	20GETLAM	#6154Ch
21PUTLAM	#61725h	21GETLAM	#6155Ch
22PUTLAM	#61735h	22GETLAM	#6156Ch

The example program QRT2 uses these combination objects to yield a somewhat more efficient program. Here, we use DOBIND instead of BIND.

```
QRT2 184 Bytes Checksum #12B1h
( %a %b %c → %root1 %root2 )
::
  0LASTOWDOB! CK3NOLASTWD                    Expect three arguments
  CK&DISPATCH1 3REAL                         Insist on three real numbers
  ::
    %0 %0                                    Placeholder values for root1 and root2
    ' :: 6GETLAM %2 %* %/ ;                  Place subroutine on the stack
                                             Temporary variable null names:
    ' NULLLAM                                  a will be in temporary variable 6
    ' NULLLAM                                  b will be in temporary variable 5
    ' NULLLAM                                  c will be in temporary variable 4
    ' NULLLAM                                  root1 will be in temporary variable 3
    ' NULLLAM                                  root2 will be in temporary variable 2
    ' NULLLAM                                  Subr will be in temporary variable 1
    SIX DOBIND                               Create temporary variable environment
    ::
      5GETLAM DUP %* 6GETLAM 4GETLAM %* %4 %* %-   Evaluate b^2 - 4ac
      DUP %0< casedrop "Complex Roots"       If <0, drop quantity, put string on
                                             stack, abandon temp env. and exit
                                             secondary
      %SQRT                                  Evaluate SQRT(b^2 - 4ac)
      5GETLAM %CHS OVER %+ 1GETLAM EVAL      Calculate first root
      3PUTLAM                                Store first root
      5GETLAM %CHS SWAP %- 1GETLAM EVAL      Calculate second root
      2PUTLAM                                Store second root
      3GETLAM                                Return first root to the user
      2GETLAM                                Return second root to the user
    ;
    ABND                                     Abandon temporary environment
  ;
;
```

As an exercise, try rewriting this example to use `CACHE` (described on page 64) instead of `DOBIND`.

3.8.3 Programming Hint for Temporary Variables

Notice that for a non-trivial program the source code can quickly turn into a blizzard of *n*`PUTLAM`'s and *n*`GETLAM`'s which become hard to read. The RPL compiler's DEFINE directive can be used to associate easier-to-remember words with objects like `17GETLAM`.

The code in QRT2.S is more efficient than the code in QRT1.S, but the code becomes less readable. When the source code is being prepared with RPLCOMP.EXE on a PC, DEFINE statements can be used to make the source code easier to manage. There are two techniques for using DEFINE with local variable names. The first is to use DEFINE to rename long variable names to short variable names (saving RAM). The second is to use DEFINE to map names directly to the `GETLAM` and `PUTLAM` combination objects. An example of the second use of DEFINE is the program QRT3.

We make an additional change to illustrate the use of `ABND`. In User-RPL, the trailing » in a program using local variables abandons the temporary environment. In System-RPL, an exit from a secondary can be coded with objects like case, but you must keep track of temporary environments yourself. In this example, there are two uses of `ABND`, one for the complex roots exit and one for the real roots exit. (Note that multiple exits from secondaries like this are prone to coding errors — be careful!)

```
QRT3 174 Bytes Checksum #6A6Bh
( %a %b %c → %root1 %root2 )
```

```
DEFINE a 6GETLAM
DEFINE b 5GETLAM
DEFINE c 4GETLAM
DEFINE root1 3GETLAM
DEFINE root1STO 3PUTLAM
DEFINE root2 2GETLAM
DEFINE root2STO 2PUTLAM
DEFINE Subr 1GETLAM
::
  0LASTOWDOB! CK3NOLASTWD          Expect three arguments
  CK&DISPATCH1 3REAL               Insist on three real numbers
  ::
    %0 %0                          Placeholder values for root1 and root2
    ' :: a %2 %* %/ ;              Place subroutine on the stack
    {                              List of temporary variable null names:
      NULLLAM                        a will be in temporary variable 6
      NULLLAM                        b will be in temporary variable 5
      NULLLAM                        c will be in temporary variable 4
      NULLLAM                        root1 will be in temporary variable 3
      NULLLAM                        root2 will be in temporary variable 2
      NULLLAM                        Subr will be in temporary variable 1
    }
    BIND                           Create temporary variable environment
    b DUP %* a c %* %4 %* %-       Evaluate b^2 - 4ac
    DUP %0< casedrop               If <0, drop quantity, put string on stack,
        :: "Complex Roots" ABND ;    abandon temp env. and exit secondary
    %SQRT                          Evaluate SQRT(b^2 - 4ac)
    b %CHS OVER %+ Subr EVAL       Calculate first root
    root1STO                       Store first root
    b %CHS SWAP %- Subr EVAL       Calculate second root
    root2STO                       Store second root
    root1                          Return first root to the user
    root2                          Return second root to the user
    ABND                           Abandon temporary environment
  ;
;
```

Notice that the use of DEFINEs makes the source code much easier to read.

3.8.4 Additional Temporary Variable Utilities

The following objects are available for working with temporary variables and environments. Some of these objects combine commonly used sequences of operations.

`1ABNDSWAP`	#62DB3h
Equivalent to `:: 1GETLAM ABND SWAP ;`	
ob → ob_{lam} ob	
`1GETABND`	#634B6h
Equivalent to `:: 1GETLAM ABND ;`	
→ ob_{lam}	
`1GETSWAP`	#62F07h
Equivalent to `:: 1GETLAM SWAP ;`	
ob → ob_{lam} ob	

`1LAMBIND`	#634CFh
Equivalent to `:: { NULLLAM } BIND ;`	
ob →	
`1NULLLAM{}`	#34D2Bh
Returns a list containing NULLLAM	
→ { NULLLAM }	
`2GETEVAL`	#632E5h
Equivalent to `:: 2GETLAM EVAL ;`	
→	
`4NULLLAM{}`	#52D26h
Returns a list containing four NULLLAMs	
→ { NULLLAM NULLLAM NULLLAM NULLLAM }	
`@LAM`	#07943h
Recalls temporary variable by name. If variable exists, the object and TRUE will be returned, otherwise FALSE will be returned.	
lam → ob_{lam} TRUE	
lam → FALSE	
`CACHE`	#610E9h
Saves n objects and n in a new temporary environment, with each temporary variable named with the provided name.	
ob_n ... ob_1 n name →	
`DUMP`	#61EA7h
The inverse of `CACHE`, but works only if NULLLAM was the name used. Forces a garbage collection.	
→ ob_n ... ob_1 n	
`DUP1LAMBIND`	#634CAh
Equivalent to `:: DUP { NULLLAM } BIND ;`	
ob → ob	
`DUP4PUTLAM`	#61610h
Equivalent to `:: DUP 4PUTLAM ;`	
ob → ob	
`DUPTEMPENV`	#61745h
Duplicates the topmost temporary environment	
→	
`GETLAMPAIR`	#617D8h
# is assumed to be 10**k*, where *k* is the index of the desired temporary variable. If $k \leq N$, where N is the number of temporary variables in the environment, the stored object, temporary variable name, and FALSE are returned. If $k > N$, then TRUE is returned.	
# → # TRUE	
# → # ob name FALSE	

3.9 Error Trapping

In User-RPL the `IFERR ... THEN ... [ELSE ...] END` structures may be used to trap errors. In System-RPL, the objects `ERRSET`, `ERRJMP`, and `ERRTRAP` provide error trapping capabilities.

In practice, the structure of an error trap is:

```
::
  ...
  ERRSET
  suspect_object
  ERRTRAP
  iferr_object
  ...
;
```

When *suspect_object* is being executed, any execution of the object `ERRJMP` will cause the rest of the *suspect_object* to be discarded and execution will resume at *iferr_object*. If no error occurs, *iferr_object* will be skipped and execution will continue with the following object.

3.9.1 Error Trapping Mechanics

When an error occurs, it is important that the system be returned to a known state for a graceful recovery. In particular, temporary environments and DoLoop environments that may have been established within the *suspect_object* must be discarded. The mechanism for this consists of a *protection word* associated with each environment which is initialized to zero when the environment is created by either `DO` or `BIND`.

When `ERRSET` is executed, the protection words for the most recently created temporary and DoLoop environments are incremented.

If `ERRJMP` (or a related object like `ABORT`) is executed, the remainder of the *suspect_object* is discarded and the protection words for the most recently created temporary and DoLoop environments are examined. If the protection word is non-zero, it is decremented. If the protection word is zero, the environment is discarded. Note that the protection word is a counter, and not a single state setting, so error traps can be nested.

`ERRTRAP` is executed only if no error occurred. When `ERRTRAP` is executed, the protection words in the topmost temporary and DoLoop environments are decremented and the *iferr_object* is skipped.

`ERRSET`	#04E5Eh
Increments topmost temporary and DoLoop protection words →	
`ERRTRAP`	#04EB8h
Decrements topmost temporary and DoLoop protection words and skips the next object →	
`ERRJMP`	#04ED1h
Generates an error →	

3.9.2 Generating an Error

In User-RPL the command `DOERR` generates an error, taking as its argument either a string, or a number specifying a message that is built into the HP 48 or an attached library. In System-RPL the actions of `DOERR` are divided into three actions:

- The object `ERRORSTO` stores a binary integer specifying a built-in message into a reserved memory location that can be read later. If the error is to be reported to the user as a string, the object `EXITMSGSTO` stores a pointer to the string into a reserved memory location and #70000h is stored to indicate a text error.
- The object `AtUserStack` declares user ownership of all stack objects.
- The object `ERRJMP` initiates the error jump itself.

For a list of error message numbers, see *Appendix A* on page 243.

The use of `AtUserStack` is unique to the User-RPL `DOERR`, and may not always be needed or appropriate for your error traps. The objects `ERRORCLR`, `ERRORSTO`, and `EXITMSGSTO` store error code information:

`ERRORCLR`	#04D33h
Clears the stored error number	
→	

`ERRORSTO`	#04D0Eh
Stores an error number	
# →	

`EXITMSGSTO`	#04E37h
Stores an error string	
$ →	

3.9.3 Handling an Error

When the *iferr_object* is executed, the temporary environments and DoLoop environments have been restored to the state prior to execution of the *suspect_object*. The *iferr_object* may need to consider side effects generated by the *suspect_object*, such as extra objects left on the stack or a system mode that has been altered.

Part of the action of an *iferr_object* is to interpret the error being handled. The objects `ERROR@` and `GETEXITMSG` may be used to recall the contents of stored error codes:

`GETEXITMSG`	#04E07h
Recalls the exit message string	
→ $	

`ERROR@`	#04CE6h
Recalls the error number	
→ #	

Example: A prototype error handler for a plotting application might wish to ignore math errors such as division by zero. The code fragment below uses `ERROR@` to recall the error number. If the error does not correspond to an anticipated error, the object `ERRJMP` is used to pass the error up to the next error handler. Error numbers from 769 to 773 are floating point errors. In this example the error is merely ignored.

```
::
  Begin_Plot_Loop
    ...
      ERRSET              Increment protection words
      ::                  The suspect_object
        Calculate_A_Point
        Plot_The_Point
      ;
    ERRTRAP
      ::                  The iferr_object
        ERROR@ DUP        Recall the error number
        769 #<            Less than 769?
        SWAP 773 #>       Greater than 773?
        OR IT ERRJMP      Pass the error along if not a floating point error
      ;
    ...
  End_Plot_Loop
;
```

3.9.4 Additional Error Objects

The following objects are also provided for error management:

`ABORT`	#04EA4h
Clears the stored error number and does `ERRJMP`	
→	

`DO$EXIT`			#15048h
Stores #70000h for the error number, stores the string message, does `AtUserStack`, then does `ERRJMP`			
$	→		
`DO#EXIT`			#1502Fh
Stores the error number, does `AtUserStack`, then does `ERRJMP`			
#	→		
`ERRBEEP`			#141E5h
Generates a standard error beep			
	→		
`ERROROUT`			#6383Ah
Stores the error number, then does `ERRJMP`			
#	→		
`JstGETTHEMSG`			#04D87h
Returns a message from a message table			
#	→	$	
`SETMEMERR`			#04FB6h
Generates `Insufficient Memory` error			
	→		
`SETSIZEERR`			#18CA2h
Generates `Bad Argument Value` error			
	→		
`SETTYPEERR`			#18CB2h
Generates `Bad Argument Type` error			
	→		
`SETSTACKERR`			#18CC2h
Generates `Too Few Arguments` error			
	→		
`SETIVLERR`			#29DFCh
Generates `Undefined Result` error			
	→		
`SETNONEXTERR`			#18C92h
Generates `Undefined Name` error			
	→		

3.10 Stack Operations

The objects listed here perform one or more stack operations. You can save code by using combination objects like `4PICKSWAP` instead of `FOUR PICK SWAP`. Some stack operations that are combined with binary integer math operations are also listed under *Binary Integers* on page 23. Some objects have the same address, such as `UNROT` and `3UNROLL`. You may use whichever name best matches your way of thinking about a procedure.

`#+ROLL`			#612DEh
ob_{m+n} ... ob_1 #m #n	→	ob_{m+n-1} ... ob_1 ob_{m+n}	
`#+UNROLL`			#6133Eh
ob_{m+n} ... ob_1 #m #n	→	ob_1 ob_{m+n} ... ob_2	
`#-ROLL`			#612CCh
ob_{m-n} ... ob_1 #m #n	→	ob_{m-n-1} ... ob_1 ob_{m-n}	
`#-UNROLL`			#6132Ch
ob_{m-n} ... ob_1 #m #n	→	ob_1 ob_{m-n} ... ob_2	
`#1+NDROP`			#62F75h
ob_{n+1} ... ob_1 #n	→		
`#1+PICK`			#611A3h
ob_{n+1} ... ob_1 #n	→	ob_{n+1} ... ob_1 ob_{n+1}	

`#1+ROLL`				#612F3h
	ob_{n+1} ... ob_1 #n	$\rightarrow$	ob_n ... ob_1 ob_{n+1}	
`#1+UNROLL`				#61353h
	ob_{n+1} ... ob_1 #n	$\rightarrow$	ob_1 ob_{n+1} ... ob_2	
`#2+PICK`				#611BEh
	ob_{n+2} ... ob_1 #n	$\rightarrow$	ob_{n+2} ... ob_1 ob_{n+2}	
`#2+ROLL`				#61318h
	ob_{n+2} ... ob_1 #n	$\rightarrow$	ob_{n+1}... ob_1 ob_{n+2}	
`#2+UNROLL`				#61365h
	ob_{n+2} ... ob_1 #n	$\rightarrow$	ob_1 ob_{n+2} ... ob_2	
`#3+PICK`				#611D2h
	ob_{n+3} ... ob_1 #n	$\rightarrow$	ob_{n+3} ... ob_1 ob_{n+3}	
`#4+PICK`				#611E1h
	ob_{n+4} ... ob_1 #n	$\rightarrow$	ob_{n+4} ... ob_1 ob_{n+4}	
`#+PICK`				#61184h
	ob_{m+n} ... ob_1#m #n	$\rightarrow$	ob_{m+n} ... ob_1 ob_{m+n}	
`10UNROLL`				#6312Dh
	ob_{10} ... ob_1	$\rightarrow$	ob_1 ob_{10} ... ob_2	
`2DROP`				#03258h
	ob_2 ob_1	$\rightarrow$		
`2DROP00`				#6254Eh
	ob_2 ob_1	$\rightarrow$	#0 #0	
`2DROPFALSE`				#62B0Bh
	ob_2 ob_1	$\rightarrow$	FALSE	
`2DUP`				#031ACh
	ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_2 ob_1	
`2DUP5ROLL`				#63C40h
	ob_3 ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_2 ob_1 ob_3	
`2DUPSWAP`				#611F9h
	ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_1 ob_2	
`2OVER`				#63FBAh
	ob_4 ob_3 ob_2 ob_1	$\rightarrow$	ob_4 ob_3 ob_2 ob_1 ob_4 ob_3	
`2SWAP`				#62001h
	ob_4 ob_3 ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_4 ob_3	
`3DROP`				#60F4Bh
	ob_3 ob_2 ob_1	$\rightarrow$		
`3PICK`				#611FEh
	ob_3 ob_2 ob_1	$\rightarrow$	ob_3 ob_2 ob_1 ob_3	
`3PICK3PICK`				#63C68h
	ob_3 ob_2 ob_1	$\rightarrow$	ob_3 ob_2 ob_1 ob_3 ob_2	
`3PICKOVER`				#630B5h
	ob_3 ob_2 ob_1	$\rightarrow$	ob_3 ob_2 ob_1 ob_3 ob_1	
`3PICKSWAP`				#62EDFh
	ob_3 ob_2 ob_1	$\rightarrow$	ob_3 ob_2 ob_3 ob_1	
`3UNROLL`				#60FACh
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_3 ob_2	
`4DROP`				#60F7Eh
	ob_4 ob_3 ob_2 ob_1	$\rightarrow$		
`4PICK`				#6121Ch
	ob_4 ob_3 ob_2 ob_1	$\rightarrow$	ob_4 ob_3 ob_2 ob_1 ob_4	
`4PICKOVER`				#630C9h
	ob_4 ob_3 ob_2 ob_1	$\rightarrow$	ob_4 ob_3 ob_2 ob_1 ob_4 ob_1	
`4PICKSWAP`				#62EF3h
	ob_4 ob_3 ob_2 ob_1	$\rightarrow$	ob_4 ob_3 ob_2 ob_4 ob_1	

`4ROLL`			#60FBBh
	$ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_3\ ob_2\ ob_1\ ob_4$
`4ROLLDROP`			#62864h
	$ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_3\ ob_2\ ob_1$
`4ROLLOVER`			#630A1h
	$ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_3\ ob_2\ ob_1\ ob_4\ ob_1$
`4ROLLROT`			#63001h
	$ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_3\ ob_1\ ob_4\ ob_2$
`4ROLLSWAP`			#62ECBh
	$ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_3\ ob_2\ ob_4\ ob_1$
`4UNROLL`			#6109Eh
	$ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_1\ ob_4\ ob_3\ ob_2$
`4UNROLL3DROP`			#6113Ch
	$ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	ob_1
`4UNROLLDUP`			#62D09h
	$ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_1\ ob_4\ ob_3\ ob_2\ ob_2$
`4UNROLLROT`			#63015h
	$ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_1\ ob_3\ ob_2\ ob_4$
`5DROP`			#60F72h
	$ob_5\ ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	
`5PICK`			#6123Ah
	$ob_5\ ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_5\ ob_4\ ob_3\ ob_2\ ob_1\ ob_5$
`5ROLL`			#60FD8h
	$ob_5\ ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_4\ ob_3\ ob_2\ ob_1\ ob_5$
`5ROLLDROP`			#62880h
	$ob_5\ ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_4\ ob_3\ ob_2\ ob_1$
`5UNROLL`			#610C4h
	$ob_5\ ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_1\ ob_5\ ob_4\ ob_3\ ob_2$
`6DROP`			#60F66h
	$ob_6\ ob_5\ ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	
`6PICK`			#6125Eh
	$ob_6\ ob_5\ ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_6\ ob_5\ ob_4\ ob_3\ ob_2\ ob_1\ ob_6$
`6ROLL`			#61002h
	$ob_6\ ob_5\ ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_5\ ob_4\ ob_3\ ob_2\ ob_1\ ob_6$
`6UNROLL`			#610FAh
	$ob_6\ ob_5\ ob_4\ ob_3\ ob_2\ ob_1$	$\rightarrow$	$ob_1\ ob_6\ ob_5\ ob_4\ ob_3\ ob_2$
`7DROP`			#60F54h
	$ob_7 \ldots ob_1$	$\rightarrow$	
`7PICK`			#61282h
	$ob_7 \ldots ob_1$	$\rightarrow$	$ob_7 \ldots ob_1\ ob_7$
`7ROLL`			#6106Bh
	$ob_7 \ldots ob_1$	$\rightarrow$	$ob_6 \ldots ob_1\ ob_7$
`7UNROLL`			#62BC4h
	$ob_7 \ldots ob_1$	$\rightarrow$	$ob_1 ob_7 \ldots ob_2$
`8PICK`			#612A9h
	$ob_8 \ldots ob_1$	$\rightarrow$	$ob_8 \ldots ob_1\ ob_8$
`8ROLL`			#6103Ch
	$ob_8 \ldots ob_1$	$\rightarrow$	$ob_7 \ldots ob_1\ ob_8$
`8UNROLL`			#63119h
	$ob_8 \ldots ob_1$	$\rightarrow$	$ob_1 ob_8 \ldots ob_2$
`DEPTH`			#0314Ch
	$ob_n \ldots ob_1$	$\rightarrow$	$ob_n \ldots ob_1$ #n
`DROP`			#03244h
	ob	$\rightarrow$	

DROPDUP	ob_2 ob_1	→	ob_2 ob_2	#627A7h
DROPFALSE	ob	→	FALSE	#6210Ch
DROPNDROP	ob_n ... ob_1 #n ob	→		#63FA6h
DROPNULL$	ob	→	NULL$	#04D3Eh
DROPONE	ob	→	#1	#62946h
DROPOVER	ob_3 ob_2 ob_1	→	ob_3 ob_2 ob_3	#63029h
DROPROT	ob_4 ob_3 ob_2 ob_1	→	ob_3 ob_2 ob_4	#62FC5h
DROPSWAP	ob_3 ob_2 ob_1	→	ob_2 ob_3	#6270Ch
DROPSWAPDROP	ob_3 ob_2 ob_1	→	ob_2	#62726h
DROPTRUE	ob	→	TRUE	#62103h
DROPZERO	ob	→	#0	#62535h
DUP	ob	→	ob ob	#03188h
DUP#1+PICK	ob_n ... ob_1 #n	→	ob_n ... ob_1 #n ob_n	#6119Eh
DUP3PICK	ob_2 ob_1	→	ob_2 ob_1 ob_1 ob_2	#611F9h
DUP4UNROLL	ob_3 ob_2 ob_1	→	ob_1 ob_3 ob_2 ob_1	#61099h
DUPDUP	ob	→	ob ob ob	#62CB9h
DUPONE	ob	→	ob ob #1	#63A9Ch
DUPPICK	ob_n ... ob_1 #n	→	ob_n ... ob_1 #n ob_{n-1}	#630DDh
DUPROLL	ob_n ... ob_1 #n	→	ob_n ob_{n-2} ... ob_1 #n ob_{n-1}	#630F1h
DUPROT	ob_2 ob_1	→	ob_1 ob_1 ob_2	#62FB1h
DUPTWO	ob	→	ob ob #2	#63AD8h
DUPUNROT	ob_2 ob_1	→	ob_1 ob_2 ob_1	#61380h
DUPZERO	ob	→	ob ob #0	#63A88h
N+1DROP	ob_{n+1} ... ob_1 #n	→		#62F75h
NDROP	ob_n ... ob_1 #n	→		#0326Eh
NDROPFALSE	ob_n ... ob_1 #n	→	FALSE	#169A5h
NDUP	ob_n ... ob_1 #n	→	ob_n ... ob_1 ob_n ... ob_1	#031D9h

Word	Stack before		Stack after	Address
`NDUPN`	ob #n	$\rightarrow$	ob ... ob #n	#5E370h
`ONEFALSE`		$\rightarrow$	#1 FALSE	#63533h
`ONESWAP`	ob	$\rightarrow$	#1 ob	#62E67h
`OVER`	ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_2	#032C2h
`OVER5PICK`	ob_4 ob_3 ob_2 ob_1	$\rightarrow$	ob_4 ob_3 ob_2 ob_1 ob_2 ob_4	#63C90h
`OVERDUP`	ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_2 ob_2	#62CCDh
`OVERSWAP`	ob_2 ob_1	$\rightarrow$	ob_2 ob_2 ob_1	#62D31h
`OVERUNROT`	ob_2 ob_1	$\rightarrow$	ob_2 ob_2 ob_1	#62D31h
`PICK`	ob_n ... ob_1 #n	$\rightarrow$	ob_n ... ob_1 ob_n	#032E2h
`ROLL`	ob_n ... ob_1 #n	$\rightarrow$	ob_{n-1} ... ob_1 ob_n	#03325h
`ROLLDROP`	ob_n ... ob_1 #n	$\rightarrow$	ob_{n-1} ... ob_1	#62F89h
`ROLLSWAP`	ob_n ... ob_1 #n	$\rightarrow$	ob_{n-1} ... ob_2 ob_n ob_1	#62D45h
`ROT`	ob_3 ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_3	#03295h
`ROT2DROP`	ob_3 ob_2 ob_1	$\rightarrow$	ob_2	#62726h
`ROT2DUP`	ob_3 ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_3 ob_1 ob_3	#62C7Dh
`ROTDROP`	ob_3 ob_2 ob_1	$\rightarrow$	ob_2 ob_1	#60F21h
`ROTDROPSWAP`	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_2	#60F0Eh
`ROTDUP`	ob_3 ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_3 ob_3	#62775h
`ROTOVER`	ob_3 ob_2 ob_1	$\rightarrow$	ob_2 ob_1 ob_3 ob_1	#62CA5h
`ROTROT2DROP`	ob_3 ob_2 ob_1	$\rightarrow$	ob_1	#6112Ah
`ROTSWAP`	ob_3ob_2 ob_1	$\rightarrow$	ob_2 ob_3 ob_1	#60EE7h
`SWAP`	ob_2 ob_1	$\rightarrow$	ob_1 ob_2	#03223h
`SWAP2DUP`	ob_2 ob_1	$\rightarrow$	ob_1 ob_2 ob_1 ob_2	#6386Ch
`SWAP3PICK`	ob_3 ob_2 ob_1	$\rightarrow$	ob_3 ob_1 ob_2 ob_3	#63C54h
`SWAP4PICK`	ob_4 ob_3 ob_2 ob_1	$\rightarrow$	ob_4 ob_3 ob_1 ob_2 ob_4	#63C7Ch
`SWAP4ROLL`	ob_4 ob_3 ob_2 ob_1	$\rightarrow$	ob_3 ob_1 ob_2 ob_4	#63C2Ch
`SWAPDROP`	ob_2 ob_1	$\rightarrow$	ob_1	#60F9Bh

SWAPDROPDUP				#62830h
	ob_2 ob_1	$\rightarrow$	ob_1 ob_1	
SWAPDROPSWAP				#6284Bh
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_3	
SWAPDROPTRUE				#21660h
	ob_2 ob_1	$\rightarrow$	ob_1 TRUE	
SWAPDUP				#62747h
	ob_2 ob_1	$\rightarrow$	ob_1 ob_2 ob_2	
SWAPONE				#63AB0h
	ob_2 ob_1	$\rightarrow$	ob_1 ob_2 #1	
SWAPOVER				#61380h
	ob_2 ob_1	$\rightarrow$	ob_1 ob_2 ob_1	
SWAPROT				#60F33h
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_2 ob_3	
SWAPTRUE				#4F1D8h
	ob_2 ob_1	$\rightarrow$	ob_1 ob_2 TRUE	
UNROLL				#0339Eh
	ob_n ... ob_1 #n	$\rightarrow$	ob_1 ob_n ... ob_2	
UNROT				#60FACh
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_3 ob_2	
UNROT2DROP				#6112Ah
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1	
UNROTDROP				#6284Bh
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_3	
UNROTDUP				#62CF5h
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_3 ob_2 ob_2	
UNROTOVER				#6308Dh
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_3 ob_2 ob_3	
UNROTSWAP				#60F33h
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_2 ob_3	
UNROTSWAPDRO				#60F0Eh
	ob_3 ob_2 ob_1	$\rightarrow$	ob_1 ob_2	
ZEROOVER				#63079h
	ob	$\rightarrow$	ob #0 ob	
ZEROSWAP				#62E3Ah
	ob	$\rightarrow$	#0 ob	
reversym				#5DE7Dh
	ob_n ... ob_1 #n	$\rightarrow$	ob_1 ... ob_n #n	

NOTE: The object `reversym` is written in System-RPL and is slow — see the program RVRSO on page 217 in *Writing Your Own Code Objects* on page 213 for an assembly language version that's much faster.

3.11 Control Structure Examples

There are an infinite number of ways to illustrate the objects and techniques that have just been described in this chapter. The first two examples provided here check an argument, loop, use `case`, and display text using objects described later in the book. The third example uses the return stack to filter a list and count the number of real number objects in the list.

You can use `SEMI` to build your own control structures in a variety of creative ways. The first two examples illustrate executing the first *n* of a series of procedures (there are many ways to approach this problem). The first approach uses a list containing all the procedures and a loop that extracts and executes the desired procedures. The second approach pushes a series of flags on the stack and uses `SEMI` to decide when to quit. The usefulness of each approach will depend on the circumstances under which it's used.

We hope these examples will stimulate some creative thinking as you consider your programming projects. Spend some time comparing these two examples. Which is faster? Why?

In the second example, why is there a `?SEMI` before the first procedure, since at this point we know that at least one procedure will be executed? Try removing it and changing the loop counter. (Hint: `DO` loops execute at least once.)

3.11.1 PLIST Example

The program PLIST executes the first *n* of a series of procedures encapsulated in a list.

```
PLIST 158.5 Bytes Checksum #F53h
( % → )
::
  0LASTOWDOB! CK1NOLASTWD          Clear saved command name, require one object
  CK&DISPATCH1 real                Require a real number
  ::
    ClrDA1IsStat RECLAIMDISP       Suspend clock, assert and clear stack display
    TURNMENUOFF                    Turn off the menu display
    SetDAsTemp                     Freeze the display when program ends
    COERCE                         Convert real number to internal binary integer
    DUP#0= caseDROP                Quit if no procedures are to be executed
    DUP FIVE #> case SETSIZEERR    Error out if more than five procedures specified
    #1+_ONE_DO (DO)                Loop from 1 to number of procedures specified
      {                            List of procedures
        :: "ONE" DISPROW1 ;          First procedure
        :: "TWO" DISPROW2 ;          Second procedure
        :: "THREE" DISPROW3 ;        Third procedure
        :: "FOUR" DISPROW4 ;         Fourth procedure
        :: "FIVE" DISPROW5 ;         Fifth procedure
      }
      INDEX@ NTHCOMPDROP           Get loop index, extract nth procedure
      EVAL                         Execute nth procedure
    LOOP                           End of loop
  ;
;
```

3.11.2 SEMI Example

The program SEMI executes the first *n* of a series of procedures separated by `SEMI` tests.

SEMI 145 Bytes Checksum #354h
(% →)

```
::
  0LASTOWDOB! CK1NOLASTWD          Clear saved command name, require one object
  CK&DISPATCH1 real                Require a real number
  ::
    ClrDA1IsStat RECLAIMDISP       Suspend clock, assert and clear stack display
    TURNMENUOFF                    Turn off the menu display
    SetDAsTemp                     Freeze the display when program ends
    COERCE                         Convert real number to internal binary integer
    DUP#0= caseDROP                Quit if no procedures are to be executed
    DUP FIVE #> case SETSIZEERR    Error out if more than five procedures specified
    TRUE SWAP                      Push TRUE on stack to signal end of process
    ZERO DO FALSE LOOP             Push n FALSE flags on the stack
    ?SEMI                          Test first flag
    "ONE" DISPROW1                 First procedure
    ?SEMI                          Test second flag
    "TWO" DISPROW2                 Second procedure
    ?SEMI                          Test third flag
    "THREE" DISPROW3               Third procedure
    ?SEMI                          Test fourth flag
    "FOUR" DISPROW4                Fourth procedure
    ?SEMI                          Test fifth flag
    "FIVE" DISPROW5                Fifth procedure
    DROP                           Drop TRUE that remains if all five procedures
                                   used
  ;
;
```

3.11.3 ticR Example

The return stack can be a handy resource for filtering through a composite object. Instead of decomposing a list on the stack and processing each object, you can put it on the return stack with `>R` and get one object at a time back for examination with `ticR`. The program RSTR uses this technique to count the number of objects in a list that are real numbers.

```
RSTR 68.5 Bytes Checksum #6340h
( {list} → %count )
```

```
::
  0LASTOWDOB! CK1NOLASTWD        Clear saved command name, require one argument
  CK&DISPATCH1 list              Require a list
  ::
    >R                           Push the list on the return stack
    %0                           The initial value of the counter
    BEGIN                        Copies I to the return stack
      RSWAP                      Swap the list to the first level
      ticR                       Pop the next object from the list

          Here, the stack is either: ( %counter object TRUE → )
                                 or: ( %counter FALSE → )

      DUP NOT ?SKIP RSWAP        If the object was not SEMI, swap the remainder of the
                                 list back

    WHILE                        If an object was found, do the WHILE clause
      :: TYPEREAL? IT %1+ ;      If the object is a real number, increment the counter
    REPEAT
  ;
;
```

Chapter 4

Objects & Object Utilities

This chapter describes several types of object and tools that manipulate them. Objects may be described as *atomic* (a single object), or *composite* (an object which is composed of one or more objects). Internal binary integers and real numbers are examples of atomic objects, and a list is an example of a composite object. This chapter covers the following object types:

Atomic Objects	Composite Objects
Bint	List
Real	Secondary
Extended Real	Symbolic
Complex	Unit
Extended Complex	
Character	
Character String	
Hex String	
Graphics Object	
Array	
Tagged	

4.1 Real & Extended Real Numbers

There are two floating point real number object types in the HP 48: *real numbers* (seen by the user), and *extended real numbers* (used internally). A real number consists of a sign, 12-digit mantissa, and a 3-digit exponent. An extended real number consists of a sign, 15-digit mantissa, and a 5-digit exponent. Exponents are stored in tens complement form. Real exponents live in the domain -500 < EEE < 500, and extended real exponents live in the domain -50000 < EEEEE < 50000.

The symbol % is used to denote a real number or an object that works with a real number. The symbol %% is used to denote an extended real number or an object that works with an extended real number. Some object names use both symbols. For instance, the object %>%% converts a real number to an extended real number.

4.1.1 Compiling Real Numbers

Real numbers can be embedded in System-RPL source code with the `%` symbol followed by a space followed by a the number. For example, the sequence `:: %RAN % .5 %* ;` returns a random number between 0 and .5.

Extended real numbers must be specified using the assembler, as RPLCOMP.EXE has trouble with them. The System-RPL code fragment below converts a real number to an extended real number, then divides that number by %% -15.3. Notice that the digits of the exponent are listed in reverse order. The last digit on the mantissa line is the sign, and is 0 for a positive number and 9 for a negative number.

```
::
  %>%%
ASSEMBLE
      CON(5)   =DOEREL
      NIBHEX   10000             Exponent
      NIBHEX   0000000000003519  Mantissa
RPL
  %%/
;
```

4.1.2 Built-In Real Numbers

The following table lists real and extended real numbers that are built into the HP 48.

Real Numbers		Extended Real Numbers	
Object	Address	Object	Address
%-MAXREAL	#2A487h	%%0	#2A4C6h
%-9	#2A42Eh	%%.1	#2A562h
%-8	#2A419h	%%.4	#2B3DDh
%-7	#2A404h	%%.5	#2A57Ch
%-6	#2A3EFh	%%1	#2A4E0h
%-5	#2A3DAh	%%2	#2A4FAh
%-4	#2A3C5h	%%3	#2A514h
%-3	#2A3B0h	%%4	#2A52Eh
%-2	#2A39Bh	%%5	#2A548h
%-1	#2A386h	%%2PI	#0F688h
%-MINREAL	#2A4B1h	%%7	#2B1FFh
%0	#2A2B4h	%%10	#2A596h
%MINREAL	#2A49Ch	%%12	#2B2DCh
%.1	#494B4h	%%60	#2B300h
%.5	#650BDh	%%PI	#2A458h
%1	#2A2C9h		
%2	#2A2DEh		
%e	#650A8h		
%3	#2A2F3h		
%PI	#2A443h		
%4	#2A308h		
%5	#2A31Dh		
%6	#2A332h		
%7	#2A347h		
%8	#2A35Ch		
%9	#2A371h		
%10	#650E7h		
%11	#1CC03h		
%12	#1CC1Dh		
%13	#1CC37h		
%14	#1CC51h		
%15	#1CC85h		
%16	#1CD3Ah		
%17	#1CD54h		
%18	#1CDF2h		
%19	#10E07h		
%20	#1CC6Bh		
%21	#1CCA4h		
%22	#1CCC3h		
%23	#1CCE2h		
%24	#1CD01h		
%25	#1CD20h		
%26	#1CD73h		
%27	#1CD8Dh		
%100	#415F1h		
%180	#650FCh		
%360	#65126h		
%MAXREAL	#2A472h		

4.1.3 Real Number Conversions

The following objects convert between real and extended real objects:

`%>%%` #2A5C1h
Converts a real number to an extended real number
% → %%

`%%>%` #2A5B0h
Converts an extended real number to a real number
%% → %

`2%>%%` #2B45Ch
Converts two real numbers to extended real numbers
% % → %% %%

`2%%>%` #2B470h
Converts two extended real numbers to real numbers
%% %% → % %

4.1.4 Real Number Functions

The following functions operate on real numbers:

`%1+` #50262h
Adds one to a real number
% → %

`%1-` #50276h
Subtracts one from a real number
% → %

`%1/` #2AAAFh
Inverse
% → %

`%10*` #62BF1h
Multiplies a real number by 10
% → %

`%ABS` #2A900h
Absolute value
% → %

`%ACOS` #2ACF1h
Arc cosine
% → %
% → C%

`%ACOSH` #2AE13h
Inverse hyperbolic cosine
% → %
% → C%

`%ALOG` #2ABBAh
Antilogarithm
% → %

`%ANGLE` #2AD38h
Angle from %x and %y (uses current angle mode)
%x %y → %

`%ASIN` #2ACC1h
Arc sine
% → %

`%ASINH` #2AE00h
Inverse hyperbolic sine
% → %

`%ATAN` #2AD21h
Arc tangent
% → %

`%ATANH`	#2AE26h
Inverse hyperbolic tangent	
% → %	
% → C%	

`%CEIL`	#2AF73h
Next greatest integer	
% → %	

`%CH`	#2AA30h
Percent change from x to y as a percentage of x	
%x %y → %	

`%CHS`	#2A920h
Change sign	
% → %	

`%COMB`	#2AE62h
Combinations of n objects taken m at a time	
%n %m → %	

`%COS`	#2AC40h
Cosine	
% → %	

`%COSH`	#2ADDAh
Hyperbolic cosine	
% → %	

`%D>R`	#2A622h
Converts degrees to radians	
% → %	

`%EXP`	#2AB2Fh
Natural exponential	
% → %	

`%EXPM1`	#2AB42h
Natural exponential minus 1	
% → %	

`%EXPONENT`	#2AE39h
Returns exponent	
% → %	

`%FACT`	#2B0C4h
Factorial or gamma function	
% → %	

`%FLOOR`	#2AF86h
Next smallest integer	
% → %	

`%FP`	#2AF4Dh
Fractional part	
% → %	

`%HMS+`	#2A6A0h
Adds in HH.MMSSs format	
% % → %	

`%HMS-`	#2A6C8h
Subtracts in HH.MMSSs format	
% % → %	

`%HMS>`	#2A68Ch
Converts a number from HH.MMSSs format to decimal hours	
% → %	

`%>HMS`	#2A673h
Converts a number from decimal hours to HH.MMSSs format	
% → %	

`%IP`	#2AF60h
Integer part	
% → %	

`%LN`	#2AB6Eh
Natural logarithm	
% → %	
% → C%	

`%LNP1`	#2ABA7h
Natural logarithm of (argument + 1)	
% → %	

`%LOG`	#2AB81h
Common logarithm	
% → %	
% → C%	

`%MANTISSA`	#2A930h
Returns mantissa	
% → %	

`%MAX`	#2A6F5h
Maximum of two numbers	
% % → %	

`%MIN`	#2A70Eh
Minimum of two numbers	
% % → %	

`%MOD`	#2ABDCh
Modulo	
% % → %	

`%NFACT`	#2AE4Ch
Factorial	
% → %	

`%NROOT`	#2AA81h
%nth root of %x	
%x %n → %	

`%OF`	#2A9C9h
Returns percentage of %x that is %y	
%x %y → %	

`%PERM`	#2AE75h
Permutations of %m items taken %n at a time	
%m %n → %	

`%POL>%REC`	#2B4BBh
Polar to rectangular conversion	
%x %y → %radius %angle	

`%R>D`	#2A655h
Radians to degrees conversion	
% → %	

`%RAN`	#2AFC2h
Generates random number in the range ($0 \leq n < 1$)	
→ %	

`%RANDOMIZE`	#2B044h
Sets the random number seed. If % is zero, the system clock is used.	
% →	

`%REC>%POL`	#2B48Eh
Rectangular to polar conversion	
%radius %angle → %x %y	

`%SGN`	#2A8D7h
Sign of a real number (-1, 0, or 1)	
% → %	

`%SIN`	#2ABEFh
Sine	
% → %	

`%SINH`	#2ADAEh
Hyperbolic sine	
% → %	

`%SPH>%REC`	#2B4F2h
Spherical to rectangular conversion	
%r %θ %ϕ → %x %y %z	

`%SQRT`	#2AB09h
Square root	
% → %	
% → C%	

`%T`	#2AA0Bh
Percent total of %x that is represented by %y	
%x %y → %	

`%TAN`	#2AC91h
Tangent	
% → %	

`%TANH`	#2ADEDh
Hyperbolic tangent	
% → %	

`%^`	#2AA70h
Exponential	
%x %y → %x^%y	

`DDAYS`	#0CC39h
Days between dates in MM.DDYYYY format (respects flag 42)	
% % → %	

`RNDXY`	#2B529h
Rounds %x to %n places	
%x %n → %	

`TRCXY`	#2B53Dh
Truncates %x to %n places	
%x %n → %	

4.1.5 Extended Real Number Functions

The following functions operate on extended real numbers:

`%%*`	#2A99Ah
Multiply	
%% %% → %%	

`%%*ROT`	#62FEDh
Multiply followed by ROT	
ob_1 ob_2 %% %% → ob_2 %% ob_1	

`%%*SWAP`	#62EA3h
Multiply followed by SWAP	
ob %% %% → %% ob	

`%%*UNROT`	#63C18h
Multiply followed by UNROT	
ob_1 ob_2 %% %% → %% ob_1 ob_2	

`%%+` #2A943h
Addition
%% %% → %%

`%%-` #2A94Fh
Subtraction
%% %% → %%

`%%/` #2A9E8h
Division
%% %% → %%

`%%^` #2AA5Fh
Exponential
%%x %%y → %%x^%%y

`%%/>%` #63B82h
Division, returns real result
%% %% → %

`%%1/` #2AA92h
Reciprocal
%% → %%

`%>%%1/` #2AA9Eh
Convert % to %%, then do reciprocal
% → %%

`%%ABS` #2A8F0h
Absolute value
%% → %%

`%%ACOSRAD` #2AD08h
Arc cosine using radians
%% → %%

`%%ANGLE` #2AD4Fh
Angle from %%x and %%y using current angle mode
%%x %%y → %%angle

`%%ANGLEDEG` #2AD6Ch
Angle from %%x and %%y using degrees
%%x %%y → %%angle

`%%ANGLERAD` #2ACD8h
Angle from %%x and %%y using radians
%%x %%y → %%angle

`%%ASINRAD` #2ACD8h
Arc sine using radians
%% → %%

`%%CHS` #2A910h
Change sign
%% → %%

`%%COS` #2AC57h
Cosine
%% → %%

`%%COSDEG` #2AC68h
Cosine using degrees
%% → %%

`%%COSH` #2ADC7h
Hyperbolic cosine
%% → %%

`%%COSRAD` #2AC78h
Cosine using radians
%% → %%

Command	Description	Input		Output	Address
`%%EXP`	Natural exponential	%%	→	%%	#2AB1Ch
`%%FLOOR`	Next smallest integer	%%	→	%%	#2AF99h
`%%H>HMS`	Decimal hours to HH.MMSSs	%%	→	%%	#2AF27h
`%%INT`	Integer part	%%	→	%%	#2AF99h
`%%LN`	Natural logarithm	%%	→	%%	#2AB5Bh
`%%LNP1`	Natural logarithm of argument plus 1	%%	→	%%	#2AB94h
`%%MAX`	Maximum of two numbers	%% %%	→	%%	#2A6DCh
`%%P>R`	Polar to rectangular conversion	%%radius %%angle	→	%%x %%y	#2B4C5h
`%%R>P`	Rectangular to polar conversion	%%x %%y	→	%%radius %%angle	#2B498h
`%%SIN`	Sine	%%	→	%%	#2AC06h
`%%SINDEG`	Sine using degrees	%%	→	%%	#2AC17h
`%%SINH`	Hyperbolic sine	%%	→	%%	#2AD95h
`%%SQRT`	Square root	%%	→	%%	#2AAEAh
`%%TANRAD`	Tangent using radians	%%	→	%%	#2ACA8h

4.2 Complex Numbers

Complex number objects contain two real number object bodies, with the same mantissa and exponent structure as real numbers. Likewise, extended complex number objects contain two extended real number object bodies.

The symbol C% is used to denote a complex number, and C%% is used to denote an extended complex number.

4.2.1 Compiling Complex Numbers

Complex numbers can be embedded in System-RPL source code with the C% symbol followed by a space followed by the real component, a space, and the imaginary component. For example, `:: ... C% 3.5 4.2 ... ;` specifies the number (3.5,4.2).

Extended complex numbers must be specified using the assembler, as RPLCOMP.EXE has trouble with them. The code fragment below shows how the extended complex number (1.25,-.83) is specified in a System-RPL source file. The prologue is followed by two extended real bodies, the first being the real part.

```
 ::
   ...
 ASSEMBLE
       CON(5)      =DOECMP
       NIBHEX      00000             Real Exponent
       NIBHEX      0000000000005210  Real Mantissa
       NIBHEX      99999             Imaginary Exponent
       NIBHEX      0000000000000389  Imaginary Mantissa
 RPL
   ...
 ;
```

4.2.2 Complex Number Conversions

The following objects convert between real, extended real, complex, and extended complex objects:

`%%>C%`	#51A07h
Converts two extended real numbers into a complex number	
$\%\%_{\mathrm{real}}\ \%\%_{\mathrm{imag}} \rightarrow$ C%	
`%>C%`	#05C27h
Converts two real numbers into a complex number	
$\%_{\mathrm{real}}\ \%_{\mathrm{imag}} \rightarrow$ C%	
`C%%>%%`	#05DBCh
Converts an extended complex number into two extended real numbers	
C%% $\rightarrow \%\%_{\mathrm{real}}\ \%\%_{\mathrm{imag}}$	
`C%%>C%`	#519F8h
Converts an extended complex number into a complex number	
C%% $\rightarrow$ C%	
`C%>%`	#05D2Ch
Converts a complex number into two real numbers	
C% $\rightarrow \%_{\mathrm{real}}\ \%_{\mathrm{imag}}$	
`C%>%%`	#519CBh
Converts a complex number into two extended real numbers	
C% $\rightarrow \%\%_{\mathrm{real}}\ \%\%_{\mathrm{imag}}$	
`C%>%%SWAP`	#519DFh
Converts a complex number into two extended real numbers, then does SWAP	
C% $\rightarrow \%\%_{\mathrm{imag}}\ \%\%_{\mathrm{real}}$	
`C>Im%`	#519B7h
Extracts the imaginary portion of a complex number	
C% $\rightarrow \%_{\mathrm{imag}}$	
`C>Re%`	#519A3h
Extracts the real portion of a complex number	
C% $\rightarrow \%_{\mathrm{real}}$	
`Re>C%`	#519A3h
Creates a complex from a real number with implied 0 imaginary part	
$\%_{\mathrm{real}} \rightarrow (\%_{\mathrm{real}},0)$	
`SWAP%>C%`	#632A9h
Does SWAP, then converts two real numbers into a complex number	
$\%_{\mathrm{imag}}\ \%_{\mathrm{real}} \rightarrow$ C%	

4.2.3 Built-In Complex Numbers

The following table lists complex and extended complex numbers that are built into the HP 48:

Object	Address
`C%-1`	#5196Ah
`C%0`	#524AFh
`C%1`	#524F7h
`C%%1`	#5193Bh

4.2.4 Complex Number Functions

The following functions operate on complex or extended complex numbers:

`C%1/`	#51EFAh
Inverse	
C% → C%	
`C%ABS`	#52062h
Returns radius from (0,0) to (x,y)	
(x,y) → %	
`C%ACOS`	#52863h
Arc cosine	
C% → C%	
`C%ACOSH`	#52836h
Hyperbolic arc cosine	
C% → C%	
`C%ALOG`	#52305h
Common antilog	
C% → C%	
`C%ARG`	#52099h
Returns angle from (x,y)	
(x,y) → %	
`C%ASIN`	#52804h
Arc sine	
C% → C%	
`C%ASINH`	#5281Dh
Hyperbolic arc sine	
C% → C%	
`C%ATAN`	#52675h
Arc tangent	
C% → C%	
`C%ATANH`	#527EBh
Hyperbolic arc tangent	
C% → C%	
`C%C^C`	#52374h
Complex number raised to complex number	
C%x C%y → C%x^C%y	
`C%C^R`	#52360h
Complex number raised to real number	
C% % → C%	
`C%CHS`	#51B70h
Change sign	
C% → C%	
`C%%CHS`	#51B91h
Change sign	
C%% → C%%	

`C%CONJ` Conjugate	C% → C%	#51BB2h
`C%%CONJ` Conjugate	C%% → C%%	#51BC1h
`C%COS` Cosine	C% → C%	#52571h
`C%COSH` Hyperbolic cosine	C% → C%	#52648h
`C%EXP` e^x	C% → C%	#52193h
`C%LN` Natural logarithm	C% → C%	#521E3h
`C%LOG` Common logarithm	C% → C%	#522BFh
`C%R^C` Real number raised to complex number	% C% → C%	#52342h
`C%SGN` Returns unit vector in the direction of z	C% → C%	#520CBh
`C%SIN` Sine	C% → C%	#52530h
`C%SINH` Hyperbolic sine	C% → C%	#5262Fh
`C%SQRT` Square root	C% → C%	#52107h
`C%TAN` Tangent	C% → C%	#525B7h
`C%TANH` Hyperbolic tangent	C% → C%	#5265Ch

4.3 Arrays

Arrays may be used to store atomic objects of a common type. Typically, arrays are used to store real and complex numbers, and many of the objects in the HP 48 manipulate real and complex arrays. Some objects work only with real or complex valued arrays, so be sure to use the correct manipulation objects. This applies especially to the MatrixWriter, which can cause the HP 48 to lose memory with arrays that are not composed of real or complex numbers.

A string array is a good place to store a large number of strings, such as prompts or error messages, in an application. Notice that while an array can be compiled (see below), and that an element can be obtained from an array (see `GETATELN` below), there is no object giving the equivalent of the User-RPL object `PUT` for an array of any object type other than real or complex numbers.

4.3.1 Compiling Arrays

The RPLCOMP.EXE compiler may be used to generate arrays of other objects, like internal binary integers or strings. For example, the code fragment below specifies an array of strings:

```
::
  ...
  ARRY [
    "Joe"
    "Fred"
    "Janet"
    "Jim"
  ]
  ...
;
```

4.3.2 Array Utilities

The objects described below may be used to work with array objects. The following notation convention applies to these descriptions:

[array]	An array of arbitrary type with one or two dimensions
[%array]	An array of real numbers with one or two dimensions
[C%array]	An array of complex numbers with one or two dimensions
[1-D array]	A vector
[2-D array]	A two dimensional array
{dims}	A list containing a bint specifying a number of elements or two bints specifying a number of rows and columns
#pos	A row-order position within an array

`ARSIZE`		#03562h
Returns the number of elements in an array		
[array]	→	#elements
`GETATELN`		#0371Dh
Returns an element from an array and TRUE if the element exists, otherwise returns FALSE		
#pos [array]	→	ob TRUE
#pos [array]	→	FALSE
`MAKEARRY`		#03442h
Creates an array with all elements equal to the specified object		
{ #rows #cols } ob	→	[array]
`MATCON`		#35CAEh
Sets all elements in an array to a real or complex number		
[%array] %	→	[%array]
[C%array] C%	→	[C%array]
`MATREDIM`		#37E0Fh
Redimensions a real or complex array. New elements are filled with %0 or C%0,0.		
[%array] {dims}	→	[%array]
[C%array] {dims}	→	[C%array]
`MATTRN`		#3811Fh
Transposes a real or complex array.		
[%array]	→	[%array]
[C%array]	→	[C%array]

`MDIMS`			#357A8h
Returns the dimensions of an array			
[1-D array]	→	#elements FALSE	
[2-D array]	→	#rows #cols TRUE	

`MDIMSDROP`			#62F9Dh
Does `MDIMS`, then `DROP`			
[1-D array]	→	#elements	
[2-D array]	→	#rows #cols	

`OVERARSIZE`			#63141h
Does `OVER`, then `ARSIZE`			
[array] ob	→	[array] ob #elements	

`PULLREALEL`			#355B8h
Returns the specified real number from a real array			
[%array] #pos	→	[%array] %	

`PULLCMPEL`			#355C8h
Returns the specified complex number from a complex array			
[C%array]	→	[C%array] C%	

`PUTEL`			#35628h
Places a real or complex number into a real or complex array at a specified location			
[%array] % #pos	→	[%array]	
[C%array] C% #pos	→	[C%array]	

`PUTREALEL`			#3566Th
Places a real number into a real array at a specified location			
[%array] % #pos	→	[%array]	

`PUTCMPEL`			#356F3h
Places a complex number into a complex array at a specified location			
[C%array] C% #pos	→	[C%array]	

4.3.3 The MatrixWriter

The MatrixWriter can be started by executing either `DoNewMatrix` to create a new array or `DoOldMatrix` to edit a array on the stack.

`DoNewMatrix`				#44C31h
Starts the MatrixWriter and creates a new array				
	→	[array]	*If terminated with* [ENTER]	
	→		*If terminated with* [CANCEL]	

`DoOldMatrix`				#44FE7h
Starts the MatrixWriter on an existing array on the stack				
[array]	→	[array] TRUE	*If terminated with* [ENTER]	
[array]	→	FALSE	*If terminated with* [CANCEL]	

4.4 Tagged Objects

Tagging an object with a meaningful label is one useful option for labeling a result being returned to the user. When accepting input from the user, it may be necessary to remove all tags from the base object before deciding if the input is valid. The objects described below facilitate these tasks.

Note that `CK&DISPATCH1` removes tags recursively as it filters user input, while `CK&DISPATCH0` does not remove tags (see *Argument Validation* on page 52).

`%>TAG`			#22618h
Tags an object with a real number			
ob %	→	tagged	

`>TAG`		#05E81h
Tags an object with a string. Has no length check (see `USER$>TAG`)		
ob $	→	tagged

`ID>TAG`		#05F2Eh
Tags an object with an a name		
ob ID	→	tagged

`STRIPTAGS`		#64775h
Removes all tags from an object		
tagged	→	ob

`STRIPTAGSl2`		#647A2h
Removes all tags from an object in level 2		
$\text{tagged}_2\ \text{ob}_1$	→	$\text{ob}_2\ \text{ob}_1$

`TAGOBS`		#647BBh
Tags one object or a series of objects		
ob $	→	tagged
$\text{ob}_1 \ldots \text{ob}_n$ { $\$_1 \ldots \$_n$)	→	$\text{tagged}_1 \ldots \text{tagged}_n$

`USER$>TAG`		#225F5h
Tags an object with a string. Issues error if string length is > 255		
ob $	→	tagged

4.5 Characters and Character Strings

There are two object types representing character information. *Character objects* (type 24) represent a single character, and *character strings* (type 2) contain one or more characters. The following objects are useful for converting to and from character objects:

`#>CHR`		#05A75h
Creates a character object with a specified character code		
#	→	chr

`CHR>#`		#05A51h
Returns a binary integer representing a character's code		
chr	→	#

`CHR>$`		#6475Ch
Converts a character object to a one character string object		
chr	→	$

4.5.1 Built-In Character Objects

The following table lists character objects that are built into the HP 48.

Num	Name	Address	Num	Name	Address
0	CHR_00	#6541Eh	85	CHR_U	#65559h
10	CHR_Newline	#6566Ah	86	CHR_V	#65560h
31	CHR_...	#65425h	87	CHR_W	#65567h
32	CHR_Space	#65686h	88	CHR_X	#6556Eh
34	CHR_DblQuote	#6542Ch	89	CHR_Y	#65575h
35	CHR_#	#65433h	90	CHR_Z	#6557Ch
40	CHR_LeftPar	#65663h	91	CHR_[	#65694h
41	CHR_RightPar	#65678h	93	CHR_]	#6569Bh
42	CHR_*	#6543Ah	95	CHR_UndScore	#6568Dh
43	CHR_+	#65441h	97	CHR_a	#65583h
44	CHR_,	#65448h	98	CHR_b	#6558Ah
45	CHR_-	#6544Th	99	CHR_c	#65591h
46	CHR_.	#65456h	100	CHR_d	#65598h
47	CHR_/	#6545Dh	101	CHR_e	#6559Fh
48	CHR_0	#65464h	102	CHR_f	#655A6h
49	CHR_1	#6546Bh	103	CHR_g	#655ADh
50	CHR_2	#65472h	104	CHR_h	#655B4h
51	CHR_3	#65479h	105	CHR_i	#655BBh
52	CHR_4	#65480h	106	CHR_j	#655C2h
53	CHR_5	#65487h	107	CHR_k	#655C9h
54	CHR_6	#6548Eh	108	CHR_i	#655D0h
55	CHR_7	#65495h	109	CHR_m	#655D7h
56	CHR_8	#6549Ch	110	CHR_n	#655DEh
57	CHR_9	#654A3h	111	CHR_o	#655E5h
58	CHR_:	#654AAh	112	CHR_p	#655ECh
59	CHR_;	#654B1h	113	CHR_q	#655F3h
60	CHR_<	#654B8h	114	CHR_r	#655FAh
61	CHR_=	#654BFh	115	CHR_s	#65601h
62	CHR_>	#654C6h	116	CHR_t	#65608h
65	CHR_A	#654CDh	117	CHR_u	#6560Th
66	CHR_B	#654D4h	118	CHR_v	#65616h
67	CHR_C	#654DBh	119	CHR_w	#6561Dh
68	CHR_D	#654E2h	120	CHR_x	#65624h
69	CHR_E	#654E9h	121	CHR_y	#6562Bh
70	CHR_F	#654F0h	122	CHR_z	#65632h
71	CHR_G	#654F7h	123	CHR_{	#656A2h
72	CHR_H	#654FEh	125	CHR_}	#656A9h
73	CHR_I	#65505h	128	CHR_Angle	#6564Eh
74	CHR_J	#6550Ch	132	CHR_Integral	#6565Ch
75	CHR_K	#65513h	133	CHR_Sigma	#6567Fh
76	CHR_L	#6551Ah	135	CHR_Pi	#65671h
77	CHR_M	#65521h	136	CHR_Deriv	#65655h
78	CHR_N	#65528h	137	CHR_<=	#656B0h
79	CHR_O	#6552Fh	138	CHR_>=	#656B7h
80	CHR_P	#65536h	139	CHR_<>	#656BEh
81	CHR_Q	#6553Dh	141	CHR_->	#65639h
82	CHR_R	#65544h	171	CHR_<<	#65640h
83	CHR_S	#6554Bh	187	CHR_>>	#65647h
84	CHR_T	#65552h			

4.5.2 Built-In String Objects

The following table lists string objects that are built into the HP 48 (not including text in message tables).

Object	Contents	Address
`$_''`	`"''"`	#6571Fh
`$_2DQ`	`""""`	#65749h
`$_::`	`"::"`	#6572Dh
`$_<<>>`	`"«»"`	#656F5h
`$_ECHO`	`"ECHO"`	#65757h
`$_EXIT`	`"EXIT"`	#65769h
`$_GRAD`	`"GRAD"`	#657A7h
`$_LRParens`	`"()"`	#6573Bh
`$_R<<`	`"R∡∡"`	#656C5h
`$_R<Z`	`"R∡Z"`	#656D5h
`$_RAD`	`"RAD"`	#65797h
`$_Undefined`	`"Undefined"`	#6577Bh
`$_XYZ`	`"XYZ"`	#656E5h
`$_[]`	`"[]"`	#65711h
`$_{}`	`"{}"`	#65703h
`NEWLINE$`	"\0A"	#65238h
`SPACE$`	`" "`	#65254h

4.5.3 String Manipulation Objects

`!append$` #62376h

String concatenation for use in low memory situations — appends directly to $\$_1$ instead of making a copy

$\$_1$ $\$_2$ → $\$_3$

`!append$SWAP` #62F2Fh

String concatenation for use in low memory situations followed by `SWAP`

ob $\$_1$ $\$_2$ → $\$_3$ ob

`#1+LAST$` #63281h

Returns the tail of a string starting one character past the location specified by #

$ # → $

`#1-SUB$` #63245h

Returns a substring after subtracting one from the bint specifying the end

$ $\#_{\text{start}}$ $\#_{\text{end}}$ → $

`#:>$` #167D8h

Converts a bint into a string followed by a colon (suitable for stack level #'s)

→ $

`#>$` #167E4h

Converts a bint into a string

→ $

`$>ID` #05B15h

Converts a string object into a name object

$ → ID

`&$` #05193h

Concatenates $\$_2$ to the end of $\$_1$

$\$_1$ $\$_2$ → $\$_3$

`&$SWAP` #63F6Ah

Concatenates $\$_2$ to the end of $\$_1$, then does `SWAP`

ob $\$_1$ $\$_2$ → $\$_3$ ob

`1_#1-SUB$` #63259h

Returns substring from 1 to #-1

$ # → $

`>H$`	#0525Bh
Prepends a character object to a string	
\$ chr → \$	
`>T$`	#052EEh
Appends a character object to a string	
\$ chr → \$	
`AND$`	#18873h
Bitwise logical AND of two strings	
$\$_1$ $\$_2$ → $\$_3$	
`Blank$`	#45676h
Creates a string of # space characters	
# → \$	
`CAR$`	#050EDh
Returns the first character of a string as a character object or an empty string if the string is empty	
\$ → chr \$ → ""	
`CDR$`	#0516Ch
Returns the string less its first character or an empty string if the string is empty	
\$ → \$ \$ → ""	
`CHR>$`	#6475Ch
Converts a character object to a one character string object	
chr → \$	
`COERCE$22`	#12770h
If a string has more than 22 characters, truncates the string to 21 characters and appends an ellipsis (...)	
\$ → \$	
`Date>d$`	#0CFD9h
Converts a real number representing a date into a string	
% → \$	
`DECOMP$`	#15B13h
Decompiles an object for the stack display using current display modes	
ob → \$	
`DROPNULL$`	#04DE3h
Drops an object from the stack and returns an empty string	
ob → NULL\$	
`DUP$>ID`	#63295h
Duplicates a string, then converts string object to name object	
\$ → \$ ID	
`DUPLEN$`	#627BBh
Duplicates a string, then returns its length	
\$ → \$ $\#_{\text{length}}$	
`DUPNULL$?`	#63209h
Returns TRUE if \$ is empty	
\$ → \$ FLAG	
`EDITDECOMP$`	#15A0Eh
Decompiles an object for editing using standard display formats	
ob → \$	
`JstGETTHEMESG`	#04D87h
Retrieves a message from the built-in message table	
# → \$	

`ID>$`	#05BE9h
Converts a name object to a string object	
ID → $	
`LAST$`	#6326Dh
Returns the last # characters in a string	
$ # → $	
`LEN$`	#05636h
Returns the number of characters in a string	
$ → #	
`NEWLINE$&$`	#63191h
Appends newline character to a string	
$ → $	
`NULL$`	#055DFh
Empty string	
→ NULL$	
`NULL$?`	#0556Fh
Returns TRUE if string is empty	
$ → FLAG	
`NULL$SWAP`	#62D59h
Swaps an empty string into level 2	
ob → NULL$ ob	
`NULL$TEMP`	#1613Fh
Empty string in TEMPOB (temporary memory)	
→ ""	
`OR$`	#18887h
Bitwise logical OR of two strings	
$\$_1$ $\$_2$ → $\$_3$	
`OVERLEN$`	#05622h
Returns the length of a string in level 2	
$ ob → $ ob $\#_{\text{length}}$	
`POS$`	#645B1h
Searches forwards for a substring within a string starting at a specified position, returning zero if the substring is not found	
$\$_{\text{search}}$ $\$_{\text{find}}$ $\#_{\text{start}}$ → $\#_{\text{position}}$	
`POS$REV`	#645BDh
Searches backwards for a substring within a string starting at a specified position, returning zero if the substring is not found	
$\$_{\text{search}}$ $\$_{\text{find}}$ $\#_{\text{start}}$ → $\#_{\text{position}}$	
`PromptIdUtil`	#49709h
Returns a string in the form "ID: object"	
ID ob → $	
`SEP$NL`	#127A7h
Separates a string at the first newline character	
$ → $\$_{\text{last}}$ $\$_{\text{first}}$	
`SUB$`	#05733h
Returns a substring	
$ $\#_{\text{start}}$ $\#_{\text{end}}$ → $	
`SUB$1#`	#30805h
Returns a bint with the value of the character at the specified position	
$ $\#_{\text{position}}$ → $\#_{\text{value}}$	
`SUB$SWAP`	#62D6Dh
Does SUB$, then SWAP	
ob $ $\#_{\text{start}}$ $\#_{\text{end}}$ → $ ob	

`SWAP&$`	#622EFh
Concatenates $1 to $2	
$\$_1\ \$_2 \rightarrow \$_3$	
`TIMESTR`	#0D304h
Returns a string time and date	
$\%_{date}\ \%_{time} \rightarrow \$$	
`TOD>t$`	#0D06Ah
Converts a real number time (24-hour format) into a 9-character string	
$\% \rightarrow \$$	
`XOR$`	#1889Bh
Bitwise logical XOR of two strings	
$\$_1\ \$_2 \rightarrow \$_3$	
`a%>$`	#162B8h
Creates a string representation of a real number using the current display format, excluding commas	
$\% \rightarrow \$$	
`a%>$,`	#162ACh
Same as `a%>$`, but includes commas if commas are part of the display format	
$\% \rightarrow \$$	
`palparse`	#238A4h
Parses a string into an object. If an error occurs, returns position of error	
$\$ \rightarrow$ ob TRUE	
$\$ \rightarrow \$\ \#_{position}$ $' FALSE	

4.6 Hex Strings

User binary integers (type 10) are implemented with hex strings. Hex strings are similar in construction to character strings, except that the length is arbitrary (character strings must have an even number of nibbles in the length of the body).

4.6.1 Hex String Conversions

The following objects convert between hex strings and other object types (respecting the user's wordsize specification).

`%>#`	#543F9h
Converts a real number to a hex string	
$\% \rightarrow$ hxs	
`HXS>%`	#5435Dh
Converts a hex string to a real number	
hxs $\rightarrow \%$	
`#>HXS`	#059CCh
Converts a bint to a hex string with a length of five nibbles	
$\# \rightarrow$ hxs	
`HXS>#`	#05A03h
Creates a bint from the lower 20 bits of a hex string	
hxs $\rightarrow \#$	
`2HXSLIST?`	#51532h
Confirms list of two hex strings, then converts to bints. Useful for validating and converting user pixel coordinates for graphics operations. Generates `Bad Argument Error` if list does not contain two hex strings.	
$\{\ hxs_1\ hxs_2\ \} \rightarrow \#_1\ \#_2$	

`HXS>$`	#54061h
Creates a string representation of a hex string using the current display mode and wordsize, then appends a letter specifying the current base mode	
hxs → $	

`hxs>$`	#540BBh
Creates a string representation of a hex string using the current display mode and wordsize	
hxs → $	

4.6.2 Wordsize Control

The user's wordsize specification can be tested or altered with the following two objects:

`WORDSIZE`	#54039h
Returns the current wordsize	
→ #	

`dostws`	#53CAAh
Stores a new value for the wordsize	
# →	

4.6.3 Basic Hex String Utilities

`&HXS`	#0518Ah
Appends hxs_2 to hxs_1	
hxs_1 hxs_2 → hxs_3	

`LENHXS`	#05616h
Returns the length (in nibbles) of a hex string	
hxs → #	

`NULLHXS`	#055D5h
Returns a null hex string	
→ NULLHXS	

`SUBHXS`	#05815h
Returns a substring	
hxs $\#_{\mathrm{start}}$ $\#_{\mathrm{end}}$ → hxs	

`HXS==HXS`	#544D9h
Returns %1 if hex strings are equal	
hxs_1 hxs_2 → %	

`HXS#HXS`	#544ECh
Returns %1 if hex strings are not equal	
hxs_1 hxs_2 → %	

`HXS<HXS`	#54552h
Returns %1 if $\mathrm{hxs}_1 < \mathrm{hxs}_2$	
hxs_1 hxs_2 → %	

`HXS<=HXS`	#5453Fh
Returns %1 if $\mathrm{hxs}_1 \leq \mathrm{hxs}_2$	
hxs_1 hxs_2 → %	

`HXS>=HXS`	#5452Ch
Returns %1 if $\mathrm{hxs}_1 \geq \mathrm{hxs}_2$	
hxs_1 hxs_2 → %	

`HXS>HXS`	#54500h
Returns %1 if $\mathrm{hxs}_1 > \mathrm{hxs}_2$	
hxs_1 hxs_2 → %	

4.6.4 Hex String Math Utilities

The following objects are the dispatchees for math operations that involve user binary integers. These objects assume that the hex strings are 64 bits or shorter. Results are returned according to the user's wordsize setting.

Name	Description	Stack	Address
`bit#%*`	Multiplies hxs by %	hxs % → hxs	#542EAh
`bit%#*`	Multiplies % by hxs	% hxs → hxs	#542D1h
`bit#%+`	Adds % to hxs	hxs % → hxs	#54349h
`bit%#+`	Adds hxs to %	% hxs → hxs	#54330h
`bit#%-`	Subtracts % from hxs	hxs % → hxs	#5431Ch
`bit%#-`	Subtracts hxs from %	% hxs → hxs	#542FEh
`bit#%/`	Divides hxs by %	hxs % → hxs	#542BDh
`bit%#/`	Divides % by hxs	% hxs → hxs	#5429Fh
`bit*`	Multiply	hxs_1 hxs_2 → hxs_3	#53ED3h
`bit+`	Add	hxs_1 hxs_2 → hxs_3	#53EA0h
`bit-`	Subtract	hxs_1 hxs_2 → hxs_3	#53EB0h
`bit/`	Divide	hxs_1 hxs_2 → hxs_3	#53F05h
`bitAND`	Bitwise logical AND	hxs_1 hxs_2 → hxs_3	#53D04h
`bitASR`	Arithmetic shift right one bit	hxs → hxs	#53E65h
`bitOR`	Bitwise logical OR	hxs_1 hxs_2 → hxs_3	#53D15h
`bitNOT`	Bitwise logical NOT	hxs → hxs	#53D4Eh
`bitRL`	Circular left shift one bit	hxs → hxs	#53E0Ch

`bitRLB`			#53E3Bh
Circular left shift one byte			
hxs	→	hxs	

`bitRR`			#53DA4h
Circular right shift one bit			
hxs	→	hxs	

`bitRRB`			#53DE1h
Circular right shift one byte			
hxs	→	hxs	

`bitSL`			#53D5Eh
Shift left one bit			
hxs	→	hxs	

`bitSLB`			#53D6Eh
Shift left one byte			
hxs	→	hxs	

`bitSR`			#53D81h
Shift right one bit			
hxs	→	hxs	

`bitSRB`			#53D91h
Shift right one byte			
hxs	→	hxs	

`bitXOR`			#53D26h
Bitwise logical XOR			
hxs_1 hxs_2	→	hxs_3	

4.7 Composite Objects

Composite objects are created from a collection of arbitrary objects. They may be created, searched, and decomposed. Lists are the most commonly used composite object in User-RPL programs, but the System-RPL objects described below also let you work with secondaries and unit objects.

4.7.1 Building Composite Objects

The following objects provide null composite objects or create composite objects.

`NULL{}`			#055E9h
A null list			
	→	NULL{}	

`{}N`			#05459h
Creates a list composed of n objects			
ob_1 ... ob_n #n	→	{ ob_1 ... ob_n }	

`ONE{}N`			#23EEDh
Creates a list containing one object			
ob	→	{ ob }	

`TWO{}N`			#631B9h
Creates a list containing two objects			
ob_1 ob_2	→	{ ob_1 ob_2 }	

`THREE{}N`			#631CDh
Creates a list containing three objects			
ob_1 ob_2 ob_3	→	{ ob_1 ob_2 ob_3 }	

`NULL::`			#055FDh
A null secondary			
	→	NULL::	

`::N`	#05445h
Creates a secondary composed of n objects	
$ob_1 \ldots ob_n$ #n → :: $ob_1 \ldots ob_n$;	

`::NEVAL`	#632D1h
Creates and then executes a secondary composed of *n* objects	
$ob_1 \ldots ob_n$ #n →	

`Ob>Seco`	#63FE7h
Creates a secondary containing one object	
ob → :: ob ;	

`2Ob>Seco`	#63FFBh
Creates a secondary containing two objects	
ob_1 ob_2 → :: ob_1 ob_2 ;	

`EXTN`	#05481h
Creates a unit object consisting of numbers, string, unit operators, and umEND (see *Unit Objects* on page 102 for more details)	
$ob_1 \ldots ob_{n-1}$ umEND #n → unit	

`SYMBN`	#0546Dh
Creates a symbolic object	
Example: `ID A ID B x+ #3 SYMBN` → `'A+B'`	
$ob_1 \ldots ob_n$ #n → symb	

4.7.2 Finding the Number of Objects in a Composite Object

The following objects return the number of objects in a composite object.

`DUPLENCOMP`	#63231h
Duplicates a composite and returns the number of constituent elements	
comp → comp #n	

`LENCOMP`	#0567Bh
Returns the number of constituent elements in a composite object	
comp → #n	

4.7.3 Adding Objects to a Composite

These objects are convenient to use but slow in execution for long lists, so caution should be exercised when using these object repetitively. The delays occur as composites are taken apart with `INNERCOMP`, objects are shuffled, and the composite is reassembled. For instance, the sequence of operations for performing `>TCOMP` is something similar to the following program fragment:

```
::
  SWAP INNERCOMP          obNEW ob1 ...  obn #N
  DUP #2+ ROLL            ob1 ...  obn #N obNEW
  SWAP #1+                ob1 ...  obn obNEW #N+1
  {}N                     { ob1 ...  obn obNEW }
;
```

`apndvarlst`	#35491h
Appends an object to a list if the object is not found within the list	
{ list } ob → { list' }	

`>HCOMP`	#052C6h
Prepends an object to a composite object	
$comp_1$ ob → $comp_2$	

`>TCOMP`	#052FAh
Appends an object to a composite object	
$comp_1$ ob → $comp_2$	

`&COMP`			#0521Fh
Concatenates two composite objects			
	comp_1 comp_2	→	comp_3

`PUTLIST`			#1DC00h
Replaces an object in a list (assumes $0 \leq i \leq n$), where n is the number of list obs			
	ob #i {list }	→	{ list' }

4.7.4 Decomposing Composite Objects

The following objects decompose a composite object into its constituent objects or extract portions of a composite. It is important to remember that when an object like `DUPINCOMP` is applied to a composite, the stack contains pointers into the original composite, not pointers to separate objects in TEMPOB. This means that as long as there is at least one pointer to an object within a composite, the entire composite is retained in TEMPOB. The object `Embedded?` can determine whether an object is embedded in a composite (see *Detecting Embedded Objects* on the next page).

`CARCOMP`			#05089h
Returns a composite's first object or a null composite if the composite is null			
	comp	→	ob
	comp	→	comp *(null composite)*

`CDRCOMP`			#05153h
Returns a composite less its first object or the composite if the composite is null			
	comp	→	comp'
	comp	→	comp *(null composite)*

`DUPINCOMP`			#631E1h
Duplicates a composite and decomposes the copy			
	comp	→	comp ob_1 ... ob_n #n

`INCOMPDROP`			#62B88h
Decomposes a composite object and drops the object count			
	comp	→	ob_1 ... ob_n

`INNERCOMP`			#054AFh
Decomposes a composite object			
	comp	→	ob_1 ... ob_n #n

`INNERDUP`			#62C41h
Decomposes a composite object and duplicates the object count			
	comp	→	ob_1 ... ob_n #n #n

`NTHCOMDDUP`			#62D1Dh
Returns two copies of the ith object in a composite (ob_i is presumed to exist)			
	comp #i	→	ob_i ob_i

`NTHCOMPDROP`			#62B9Ch
Returns the ith object in a composite (ob_i is presumed to exist)			
	comp #i	→	ob_i

`NTHELCOMP`			#056B6h
Returns the ith object in a composite and TRUE or FALSE if there are not at least i elements in the composite			
	comp #i	→	ob_iTRUE
	comp #i	→	FALSE

`SUBCOMP`			#05821h
Returns a subcomposite. Indices out of range are set to composite bounds			
	comp $\#_\text{start}$ $\#_\text{end}$	→	comp'

`SWAPINCOMP`			#631F5h
Does SWAP, then decomposes a composite			
	comp obj	→	obj ob_1 ... ob_n #n

4.7.5 Searching Composite Objects

The object POSCOMP is the generalized tool for searching through a composite object for an object that satisfies some comparison with a supplied object. The following program fragment indicates the position in a composite of the first binary integer greater than #5:

```
::
  ...                            ( (list} )
  FIVE ' #> POSCOMP              (   #pos )
  ...
;
```

The objects EQUALPOSCOMP and NTHOF supply the predicate EQUAL to POSCOMP, simplifying some search procedures.

EQUALPOSCOMP	#644A3h
Returns the position of the first object in a composite equal to an object. If the object is not found, zero is returned.	
comp ob → #pos	
matchob?	#643EFh
Returns TRUE if ob is equal to any object within a composite, or ob and FALSE if not.	
ob comp → ob FALSE	
ob comp → TRUE	
NTHOF	#644BCh
Returns the position of the first object in a composite equal to an object. If the object is not found, zero is returned.	
ob comp → #pos	
POSCOMP	#64426h
Returns the position of the first object in a composite that satisfies a test with the supplied predicate and an object. If the object is not found, zero is returned.	
comp ob pred → #pos	

4.7.6 Detecting Embedded Objects

As mentioned above, an object on the stack may be contained within a composite. The object Embedded? may be used to detect this case, and CKREF can be used to check all references to an object.

CKREF	#37B44h
Creates a unique copy of an object if it is referenced or embedded in any composite object	
ob → ob	
Embedded?	#64127h
Returns TRUE if ob_2 is embedded in or is the same as ob_1	
ob_1 ob_2 → FLAG	

4.8 Unit Objects

Unit objects evolved from representing integer powers in the HP 48S/SX to real powers in the HP 48G/GX. This can be quickly demonstrated by comparing using the User-RPL function UBASE and the System-RPL object U>NCQ on the S and G series:

	HP 48S/SX	HP 48G/GX
Object	1_m^2.3/s^3.7	1_m^2.3/s^3.7
UBASE	1_m^2/s^4	1_m^2.3/s^3.7
U>NCQ	%%1 %%1 HXS 10 002000CF00000000	%%1 %%1 [%0 %2.3 %0 %-3.7 %0 %0 %0 %0 %0 %0]

The object U>NCQ is used to break apart a unit object into a number part, conversion factor, and unit quantity vector. In the S series, the unit quantities were expressed as 8 signed 8-bit quantities in a hex string. Negative unit quantities indicate units in the denominator. In the G series, the unit quantities are expressed as a 10 element real vector.

4.8.1 Dimensional Consistency

If two unit objects are dimensionally consistent, their unit quantity vectors will be equal. The unit quantity vector is formatted as follows:

Element	Quantity	Base Unit
1	mass	kilogram
2	length	meter
3	electric current	ampere
4	time	second
5	thermodynamic temperature	kelvin
6	luminous intensity	candela
7	amount of substance	mole
8	plane angle	radian
9	solid angle	steradian
10	unspecified (1_?)	

The following code fragment checks two objects for dimensional consistency, returning the system flags TRUE or FALSE:

```
:: U>NCQ ROTROT2DROP SWAP U>NCQ ROTROT2DROP EQUAL ;
```

4.8.2 Building and Decomposing Unit Objects

Unit objects are composite objects that can be broken apart with INNERCOMP and assembled with EXTN. Extending the previous example to use *km* instead of *m*, apply INNERCOMP to 1_km^2.3/s^3.7:

:: 1_km^2.3/s^3.7 INNERCOMP ; → %1 "k" "m" umP %2.3 um^ "s" %3.7 um^ um/ umEND ELEVEN

Notice that the object is constructed much the same way as an RPN expression, with the provision that umEND be the last object. If you're viewing these objects with tools like SSTK in Jazz, you'll notice that unit operators (like um/) are decompiled as {} in User-RPL. These unit operators found within a unit object are different from objects that manipulate unit objects, such as UM+, UM-, etc.

Unit Operator	Purpose	Address
um*	Multiply operator	#10B5Eh
um/	Divide operator	#10B68h
um^	Power operator	#10B72h
umP	Prefix operator	#10B7Ch
umEND	End of unit object	#10B86h

The System-RPL objects UM>U and UMU> are useful for many tasks. UMU> breaks a unit object into a number and normalized unit part, while UM>U replaces the number part of a unit object (useful when returning a unit result).

4.8.3 Unit Object Utilities

The following objects operate on unit objects.

`EXTN`	#05481h
Assembles a unit object consisting of numbers, string, unit operators, and umEND	
ob_{n-1} ... ob_1 umEND #n → unit	

`UM%`	#0FBABh
Returns a percentage of a unit quantity	
unit %percentage → unit	

`UM%CH`	#0FC3Ch
Returns the percent difference between two unit quantities	
unit_1 unit_2 → %	

`UM%T`	#0FCCDh
Returns the percentage fraction of unit_1 that is unit_2	
unit_1 unit_2 → %	

`UM*`	#0F792h
Unit multiply	
unit unit → unit	

`UM+`	#0F6A2h
Unit addition	
unit unit → unit	

`UM-`	#0F774h
Unit subtraction	
unit unit → unit	

`UM/`	#0F823h
Unit division	
unit unit → unit	

`UM>U`	#0F33Ah
Replaces the number part of a unit object	
% unit → unit	

`MABS`	#0F5FCh
Absolute value	
unit → unit	

`UMCEIL`	#0FD36h
Next greatest integer	
unit → unit	

`UMCHS`	#0F615h
Change sign	
unit → unit	

`UMCONV`	#0F371h
Unit conversion — converts unit_1 to unit_2 units	
unit_1 unit_2 → unit_1'	

`UMCOS`	#0F660h
Cosine	
unit → %	

`UMFLOOR`	#0FD22h
Next smallest integer	
unit → unit	

`UMFP`	#0FD0Eh
Fractional part	
unit → unit	

`UMIP`	#0FCFAh
Integer part	
unit → unit	

`UMMAX`	#OFB6Fh
Maximum of two unit quantities	
unit_1 unit_2 → unit	

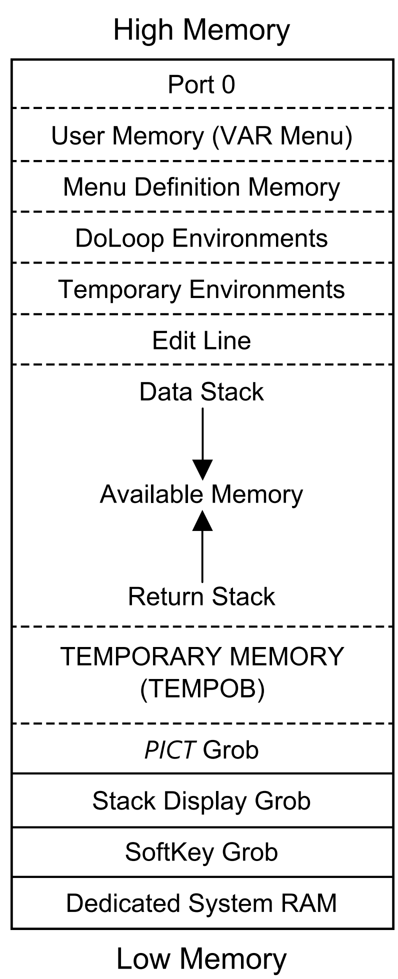
`UMMIN`			#0FB8Dh
Minimum of two unit quantities			
$unit_1$ $unit_2$	→	unit	
`UMRND`			#0FD68h
Round to specified number of places			
unit %places	→	unit	
`UMSI`			#0F945h
Converts unit quantity to SI units			
unit	→	unit	
`UMSIGN`			#0FCE6h
Returns sign (-1, 0, or 1) of unit quantity			
unit	→	%	
`UMSIN`			#0F62Eh
Sine			
unit	→	%	
`UMSQ`			#0F913h
Square			
unit	→	unit	
`UMSQRT`			#0F29Ch
Square root			
unit	→	unit	
`UMTAN`			#0F674h
Tangent			
unit	→	%	
`UMTRC`			#0FD8Bh
Truncate to specified number of places			
unit %places	→	unit	
`UMU>`			#0F34Eh
Returns number and normalized unit parts of a unit object			
unit	→	% unit'	
`UMXROOT`			#0F8FAh
Returns $unit_x$th root of $unit_y$			
$unit_x$ $unit_y$	→	unit	
`UNIT>$`			#0F218h
Decompiles a unit object			
unit	→	$	

Chapter 5

Memory Utilities

The HOME directory and its subdirectories are collectively known as USEROB, which is different from the temporary memory (TEMPOB). In TEMPOB, objects live briefly, and are discarded when memory is low and no pointers refer to them. In USEROB, an object exists until purged by a user command.

The objects described in this chapter provide some of the basic utilities for dealing with input from the user, results returned to the user, and directories. An important convention in the HP 48 is the sanctity of variables stored in user memory. Some operations, like `GROB!`, don't care where an object resides. It's therefore possible to alter a user's input arguments instead of providing a unique result. Unless there is a specific design intent, an application should not change the directory pointed to by the VAR menu when the application begins.



5.1 Name Objects

In this chapter, "ID" and "lam" refer to global and local variable name objects. The following objects convert between strings and name objects:

`$>ID`	#05B15h
Converts a string object into a name object	
$ → ID	

`DUP$>ID`	#63295h
Duplicates a string, then converts string object to name object	
$ → $ ID	

`ID>$`	#05BE9h
Converts a name object to a string object	
ID → $	

5.2 User Variables

Evaluating a user variable is just as straightforward in System-RPL as in User-RPL — just specify the name:

```
:: ... ID X ... ;
```

Since any object can be in X, or X may not exist, you might want to exercise some caution. This is part of the reason the HP 48 is criticized for being slow in some areas, especially with respect to the plotting system. When a plot is drawn, the contents of PPAR, the equation, and related variables must be validated before the plot gets underway. Since the user can provide a program for an equation definition, further checks are required to make sure the program will not inflict untoward damage. If you're at all concerned about these issues, recall the contents of the variable before evaluating.

`CREATE`	#08696h
Creates a variable in the current directory (does not check for unique name)	
ob ID →	

`?PURGE_HERE`	#1854Fh
Purges specified variable only if it exists in the current directory and does not contain a non-empty directory, otherwise generates `Non-empty Directory` error	
ID →	

`PURGE`	#08C27h
Purges the specified variable. Do *not* purge a non-empty directory with this object — use XEQPGDIR instead.	
ID →	

`@`			#0797Bh
Recalls the contents of a global or temporary variable. For global variables, begins at the current directory and searches up through HOME			
ID	→	ob TRUE	*Global variable exists*
ID	→	FALSE	*Global variable nonexistent*
lam	→	ob TRUE	*Temporary variable exists*
lam	→	FALSE	*Temporary variable nonexistent*

`Sys@`			#2EA6Ah
Recalls the contents of a global variable from HOME directory			
ID	→	ob TRUE	*Global variable exists*
ID	→	FALSE	*Global variable nonexistent*

`SAFE@`			#62A34h
Recalls the contents of a global or temporary variable. For global variables, begins at the current directory and searches up through HOME. ROM bodies are converted to XLIB names.			
ID	→	ob TRUE	*Global variable exists*
ID	→	FALSE	*Global variable nonexistent*
lam	→	ob TRUE	*Temporary variable exists*
lam	→	FALSE	*Temporary variable nonexistent*

`SAFE@_HERE`			#1853Bh
Recalls the contents of a global or temporary variable. For global variables, recalls only from the current directory. ROM bodies are converted to XLIB names.			
ID	→	ob TRUE	*Global variable exists*
ID	→	FALSE	*Global variable nonexistent*
lam	→	ob TRUE	*Temporary variable exists*
lam	→	FALSE	*Temporary variable nonexistent*

`SAFESTO`		#07D27h
Stores an object in the current directory. If the object is to be stored in a global variable and is referenced, a copy is left in temporary memory and all references are adjusted to point to the copy. Searches current and then parent directories for the global variable, replacing the contents if found, otherwise creates variable in the current directory. ROM bodies are converted to XLIB names.		
ob lam	→	
ob ID	→	

`STO`		#07D27h
Stores an object in the current directory. If the object is to be stored in a global variable and is referenced, a copy is left in tempob and all references are adjusted to point to the copy. Searches current and then parent directories for the global variable, replacing the contents if found, otherwise creates variable in the current directory.		
ob lam	→	
ob ID	→	

`SysSTO`		#2E9E6h
Stores an object in HOME		
ob ID	→	

`XEQSTOID`		#18513h
Stores an object in the current directory. If the object is to be stored in a global variable and is referenced, a copy is left in temporary memory and all references are adjusted to point to the copy. Will not overwrite a directory. This does the work for the user command STO.		
ob lam	→	
ob ID	→	

5.3 Directory Utilities

A directory is an object, but you should note that directories are *not* composite objects. To be used, a directory must be “rooted”, meaning it must be a subdirectory of the permanent HOME directory. When the HP 48 is first turned on, the HOME directory is established, and a pointer called CONTEXT refers to this HOME directory. Subdirectories are said to be “rooted” in their parent directory. As the directory structure is traversed, the CONTEXT pointer is updated to point to subdirectories within HOME. CONTEXT should *never* point to an unrooted directory, and no pointer should *ever* point within an unrooted directory, because the garbage-collection system isn’t designed to traverse a directory in TEMPOB.

`CONTEXT!`	#08D08h
Stores a pointer to a rooted directory in CONTEXT, defining the current directory	
directory →	
`CONTEXT@`	#08D5Ah
Recalls the CONTEXT pointer	
→ directory	
`CREATEDIR`	#184E1h
Creates a directory in the current directory	
ID →	
`DOVARS`	#18779h
Returns a list of the variables in the current directory	
→ { ID_1 ... ID_n }	
`PATHDIR`	#1848Ch
Returns a list describing the path from HOME to the current directory	
→ { HOME ID ID ... }	
`SYSCONTEXT`	#08D92h
Stores the HOME directory pointer into CONTEXT	
→	
`UPDIR`	#1A16Fh
Makes the parent directory the current directory	
→	
`XEQORDER`	#20FF2h
Asserts the order of IDs in the current directory	
{ ID_1 ... ID_n } →	
`XEQPGDIR`	#18595h
Purges a directory	
ID →	

The hidden directory is a null-named directory at the end of the HOME directory, and contains user key definitions and alarm information. Applications that use this directory need to either clean up after themselves or provide a user command to clear stored information.

`PuHiddenVar`	#6408Ch
Purges the specified variable in the hidden directory	
ID →	
`RclHiddenVar`	#64023h
Recalls a hidden variable using `@`	
ID → ob	
`StoHiddenVar`	#64078h
Stores an object in the hidden directory using `STO`	
ob ID →	

5.4 Temporary Memory

The data stack in the HP 48 is actually a stack of pointers which refer to objects elsewhere in memory. Temporary memory is the calculator's "scratchpad". All objects that are not stored in a port or in a user variable reside in temporary memory. Many of the objects described in this book require temporary memory to construct intermediate objects or new objects returned as results to the stack.

5.4.1 Use of Temporary Memory

To understand temporary memory a little more, consider what happens when two math operations are performed. Enter the numbers 1.5 and 2.6 on the stack. These numbers now reside in temporary memory, referred to by pointers on the data stack. When the numbers are added, the result, 4.1, is a number in temporary memory referenced by a pointer in level 1 of the data stack. The objects 1.5 and 2.6 remain in temporary memory, referenced by pointers that save the Last Arguments.

Now add 2.8 to the result in level 1. The level 1 pointer on the data stack refers to the object 6.9 in temporary memory. The last arguments pointers now refer to the objects 2.8 and 4.1, and the objects 1.5 and 2.6 are no longer referenced.

The object `TOTEMPOB` may be used to create a new copy of an object in temporary memory, whose only reference is on the data stack. In general, the system will perform an automatic `TOTEMPOB` where it makes sense. For instance, if you recall the contents of a variable to the stack and press [EDIT], the object will be copied to temporary memory before editing begins.

Sometimes you may want to "free" an object that was extracted from a list. Consider the following User-RPL program:

```
« { "AB" "CD" "EF" } 2 GET »
```

Level 1 of the data stack contains a pointer into the list, which still resides in temporary memory. Executing `NEWOB` now would create the unique object "CD" in temporary memory, and release the list for garbage collection. (*Note:* set the Last Arguments flag (-55) to prevent the list from being referenced as a last argument.)

The following objects are useful for checking references to objects and their locations.

`CKREF`	#37B44h
Creates a unique copy of an object if it's referenced, embedded, or in USEROB.	
ob → ob	

`INTEMNOTREF?`	#06B4Eh
Returns TRUE if ob is in TEMPOB, and not referenced or embedded	
ob → ob FLAG	

`SWAPCKREF`	#63F7Eh
Swaps objects, then does `CKREF`	
ob_1 ob_2 → ob_2 ob_1	

`TOTEMPOB`	#06657h
Creates a unique copy of an object in TEMPOB	
ob → ob	

5.4.2 Garbage Collection

From time to time the HP 48 will "hesitate" during an operation. This hesitation is usually caused by the removal of objects in temporary memory which are no longer being used. Objects which are no longer referenced continue to accumulate in temporary memory until memory has been filled. When memory is full, the calculator scans the objects in temporary memory, deleting those without references to them. This process, known as "garbage collection", is similar in concept to garbage collection in LISP.

A large number of pointers on the stack that point to temporary memory can slow down the garbage collection process to an uncomfortable degree. This occurs when there are a large number of objects on the stack, or an object has been extracted from a large list. A worst case scenario occurs when a list that has been stored in a local variable has been broken out onto the stack using the User-RPL command `OBJ→` or `INNERCOMP` (see *Composite Objects* on page 99). In this case, the time required for garbage collection increases roughly with the square of the number of objects that were in the list. List operations can be optimized by storing the lists in global variables, effectively moving the operations from temporary memory to user memory.

`GARBAGE`	#05F42h
Performs a garbage collection	
→	

5.5 Memory Utilities

`MEM`	#05F61h
Returns the number of nibbles of free memory. Note that you may wish to collect garbage first to get an accurate measure of available memory.	
→ #	
`OCRC`	#05944h
Returns the size of an object in nibbles as a bint and the object's checksum as a hex string	
→ #size hxs_checksum	
`OCRC%`	#1A1FCh
Returns the size of an object in bytes as a real and the object's checksum as a hex string	
→ %size hxs_checksum	
`getnibs`	#6595Ah
Replaces hex string body with data from memory at the specified address	
hxs_data hxs_address → hxs_data'	
`putnibs`	#6594Eh
Replaces memory data at the specified address with body of data hex string	
hxs_data hxs_address →	

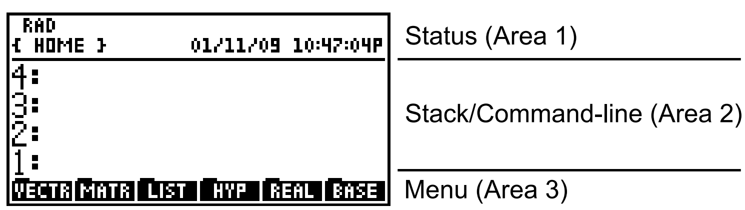
Chapter 6

Graphics, Text, and the LCD

Many people turn to System-RPL for additional control over the HP 48 display. While User-RPL graphics resources generally work with the built-in graphics object *PICT* and do not work with the stack display, System-RPL routines have fewer restrictions. This chapter will introduce the organization of the display and some basic tools for manipulating graphics objects and display memory.

6.1 LCD Display Regions

When the HP 48 is displaying the stack during normal calculations, the LCD is divided into three regions, each having display memory and objects associated with them to control display refresh.



The status area and the stack/command line area are displayed using the stack grob (ABUFF). The menu area is displayed using the menu grob (HARDBUFF2). The object `SysDisplay` updates the entire display:

`SysDisplay`	#386A1h
Displays the status, stack, and menu areas	
$\rightarrow$	

The User-RPL `FREEZE` command provides a basic way to prevent one or more of these regions from being updated when a program halts for input or terminates. There are many System-RPL objects and flags associated with these regions that perform similar tasks. Here we present a subset of these objects that should suit many applications.

6.1.1 Status Area Control

The status area is 16 pixel rows high. Two objects are of interest for the status area. `ClrDA1IsStat` suspends the clock display (this is safe to use whether or not the clock is being displayed). `SetDA1Temp` "freezes" the status area after your application halts for a prompt or terminates.

`ClrDA1IsStat`	#39531h
Suspends the ticking clock display	
$\rightarrow$	

`SetDA1Temp`	#3902Ch
Signals that the status area should not be redrawn	
→	

`SetDA1Bad`	#3947Bh
Signals that the status area should be redrawn	
→	

`DispStatus`	#395BAh
Draws the status area	
→	

`?DispStatus`	#3959Ch
If no keys are in the keybuffer, draws the status area, otherwise does not draw the display area and executes `SetDA1Bad`	
→	

6.1.2 Stack Area Control

The stack/command-line area is 40 pixel rows, and is actually divided into two sub-regions named 2a and 2b. The command line is the main portion of the HP 48 that recognizes the two sub-regions. Region 2a displays the stack, and region 2b displays the command line. Either area can be null, but in principle they both exist at all times. The object `SetDA2OKTemp` signals that neither display area 2a or 2b should be redrawn.

`SetDA2OKTemp`	#39207h
Signals that the stack/command line areas (2a and 2b) should not be redrawn	
→	

`SetDA2aTemp`	#39045h
Signals that the stack area (2a) should not be redrawn	
→	

`SetDA2bTemp`	#39059h
Signals that the command line area (2b) should not be redrawn	
→	

`SetDA2aBad`	#394A5h
Signals that the stack area (2a) should be redrawn	
→	

`SetDA2bBad`	#394CFh
Signals that the command line area (2b) should be redrawn	
→	

`?DispStack`	#39B85h
If no keys are in the keybuffer, draws the stack area, otherwise does not draw the stack area and executes `SetDA2aBad`	
→	

`DispEditLine`	#3A00Dh
Displays the edit line	
→	

6.1.3 Menu Area Control

The menu area occupies the bottom 8 pixel rows of the display. The menu area can be frozen with the object `SetDA3Temp`. The current menu definition can be displayed with either of the `DispMenu` objects (see also *Menu Utilities* on page 164).

`DispMenu`	#3A1E8h
Displays the current menu and freezes the menu display line	
→	

`DispMenu.1`	#3A1FCh
Displays the current menu	
→	

`?DispMenu`	#3A1CAh
If no keys are in the keybuffer, draws the menu area, otherwise does not draw the menu area and executes `SetDA3Bad`	
→	

`SetDA3Temp`	#39072h
Signals that the menu should not be redrawn	
→	

`SetDA3Bad`	#394F9h
Signals that the menu should be redrawn	
→	

6.1.4 Combined Area Controls

The object `ClrDAsOK` signals that the entire display should be redrawn when the application terminates. Conversely, the object `SetDAsTemp` signals that no part of the display should be redrawn (the same as `7 FREEZE` in User-RPL).

`ClrDAsOK`	#39144h
Signals entire LCD should be redrawn	
→	

`SetDA12Temp`	#3921Bh
Signals that only the menu area should be redrawn	
→	

`SetDAsTemp`	#3922Fh
Signals that no part of the LCD should be redrawn	
→	

6.2 Basic Display Memory Principles

There are three reserved graphics objects (grobs) in the HP 48: the stack grob, the menu grob, and the graphics grob (*PICT*). The HP 48's LCD always displays either the stack grob or *PICT*; the menu grob is optional in either case.

Applications wishing to be compatible with both the S and G series of the HP 48 should avoid using direct RAM addresses to refer to these grobs, since RAM was relocated for the G series. Built-in objects described in the next three subsections provide reliable pointers to these grobs.

6.2.1 The Current Display Grob

The object `HARDBUFF` returns a pointer to the currently displayed stack or *PICT* grob to the data stack:

`HARDBUFF`	#12635h
Returns the currently displayed stack or graphics grob	
→ grob	

The following objects clear all or part of the HARDBUFF grob:

`BLANKIT`	#126DFh
Clears #rows starting at the specified row	
$\#\text{row}_{\text{start}}$ #rows →	

`BlankDA12`		#3A578h
Clears rows 0 – 56		
	→	

`BlankDA1`		#3A546h
Clears rows 0 – 16		
	→	

`BlankDA2`		#3A55Fh
Clears rows 16 – 40		
	→	

`CLEARVDISP`		#134AEh
Clears all of HARDBUFF		
	→	

`Clr16`		#0E06Fh
Clears the first 16 rows		
	→	

`Clr8`		#0E083h
Clears the first 8 rows		
	→	

`Clr8-15`		#0E097h
Clears rows 8 – 15		
	→	

6.2.2 The Stack Grob

The stack display is nominally 131x56 pixels, but may be enlarged and scrolled. The object `ABUFF` puts a pointer to the stack display grob on the data stack. The object `TOADISP` switches the LCD display to the stack grob.

`ABUFF`		#12655h
Returns the stack grob		
	→ grob	

`DOCLLCD`		#5046Ah
Clears the stack grob		
	→	

`DOLCD>`		#503D4h
Returns a grob with the first 56 rows of ABUFF and a copy of the menu area at the bottom (just like the LCD)		
	→ grob	

`DO>LCD`		#50438h
Stores a grob into the upper-left corner of ABUFF		
	grob →	

`TOADISP`		#1314Dh
Displays the stack grob		
	→	

The stack display is often used by applications or games which do not wish to disturb *PICT*. The Equation Writer, MatrixWriter, and Minehunt game all use the stack display. Two objects which are useful for claiming the stack display for an application are `RECLAIMDISP` and `ClrDA1IsStat`:

`RECLAIMDISP`		#130ACh
Switches to stack display, clears, unscrolls, and resizes to default size (131x56)		
	→	

`ClrDA1IsStat`		#39531h
Disables the ticking clock display		
	→	

6.2.3 The Graphics Grob

The graphics grob (*PICT*) is nominally 131x64 pixels, but may be enlarged and scrolled. The object `GBUFF` puts a pointer to the graphics grob on the data stack. The object `TOGDISP` switches the LCD display to the graphics grob.

`GBUFF`	#12665h
Returns the graphics grob	
→ grob	

`GBUFFGROBDIM`	#5187Fh
Returns the dimensions of the graphics grob (*PICT*)	
→ #height #width	

`GROB>GDISP`	#12F94h
Stores a grob into GBUFF	
grob →	

`MAKEPICT#`	#4B323h
Replaces the graphics grob with a blank grob of specified dimensions.	
#width #height →	
Note: `MAKEPICT#` will not create a graphics grob less than 64 rows high or 131 columns wide.	

`TOGDISP`	#13135h
Displays the graphics grob (*PICT*)	
→	

`WINDOW#`	#4F052h
Displays the graphics grob (*PICT*) at the specified window coordinates. This is the object that does the work for PVIEW with pixel coordinate parameters.	
#x #y →	

6.2.4 Verifying Display Grob Height

To make sure that that either ABUFF or GBUFF are at least 64 rows high, use the object `CHECKHEIGHT`.

`CHECKHEIGHT`	#511E3h
Force either ABUFF or GBUFF to be at least 64 rows high	
grob #current_grob_height →	

Note: `CHECKHEIGHT` only works for ABUFF and GBUFF!

Example: To ensure that the stack grob is at least 64 rows high, execute the following fragment:

```
::
  ABUFF             Pointer to the stack grob
  DUPGROBDIM DROP   Height of the stack grob
  CHECKHEIGHT       Ensures stack grob is at least 64 rows high
;
```

6.2.5 Enlarging ABUFF or GBUFF

The following objects may be used to enlarge either the stack grob or the graphics grob. They *will not* work for any other grob.

`HEIGHTENGROB`	#12DD1h
Adds blank rows to the specified display grob	
grob #rows →	

`WIDENGROB`	#12BB7h
Adds blank columns to the specified display grob	
grob #cols →	

6.2.6 Scrolling ABUFF or GBUFF

If either the stack or graphics grob are larger than the size of the LCD, they may be scrolled. You can track the location of the LCD "window" into the grob by testing/setting the upper left "window" coordinates. The object `WINDOWXY` sets these coordinates, and the object `WINDOWCORNER` returns these coordinates.

`WINDOWCORNER`	#137B6h
Returns the current window coordinates	
→ #x #y	
`WINDOWXY`	#13679h
Sets the window coordinates	
#y #x →	

The following objects may be used for scrolling the display. A nice example of their use is the program `SCROLL.S`, included with the HP tools and documentation.

`JUMPBOT`	#516AEh
Move the window to the bottom edge of the grob	
→	
`JUMPLEFT`	#516E5h
Move the window to the left edge of the grob	
→	
`JUMPRIGHT`	#51703h
Move the window to the right edge of the grob	
→	
`JUMPTOP`	#51690h
Move the window to the top edge of the grob	
→	
`SCROLLDOWN`	#4D16Eh
Scroll the window down one pixel with repeat (tied to down-arrow key)	
→	
`SCROLLLEFT`	#4D150h
Scroll the window left one pixel with repeat (tied to left-arrow key)	
→	
`SCROLLRIGHT`	#4D18Ch
Scroll the window right one pixel with repeat (tied to right-arrow key)	
→	
`SCROLLUP`	#4D132h
Scroll the window up one pixel with repeat (tied to up-arrow key)	
→	
`WINDOWDOWN`	#13220h
Scroll the window down one pixel	
→	
`WINDOWLEFT`	#134E4h
Scroll the window left one pixel	
→	
`WINDOWRIGHT`	#1357Fh
Scroll the window right one pixel	
→	
`WINDOWUP`	#131C8h
Scroll the window up one pixel	
→	

6.2.7 The Menu Grob

The menu display is a fixed 131x8 pixel grob. The object HARDBUFF2 puts a pointer to the menu display grob on the data stack. The objects TURNMENUON, TURNMENUOFF, and MENUOFF? control and test the display of the menu grob. Note that when TURNMENUOFF is used to turn off the menu display, the stack display (or graphics display) grob will be enlarged from 56 to 64 rows. The object LINECHANGE does the work for TURNMENUON and TURNMENUOFF.

CLEARMENU			#51125h
Clears the menu grob			
	→		
DispMenu			#3A1E8h
Displays the current menu and freezes the menu display line (SetDA3Valid)			
	→		
DispMenu.1			#3A1FCh
Displays the current menu			
	→		
HARDBUFF2			#12645h
Returns the menu grob			
	→	grob	
LINECHANGE			#4E37Eh
Sets the display pixel row upon which to begin displaying HARDBUFF2. Valid values are from 55d (menu on) to 63d (menu off).			
#row	→	grob	
MENUOFF?			#4E360h
Returns TRUE if the menu is not displayed			
	→	FLAG	
TURNMENUOFF			#4E2CFh
Turns off the menu display			
	→		
TURNMENUON			#4E347h
Turns on the menu display			
	→		

In the example *Rolling the Menu Display* below, the object LINECHANGE will be used to show how the menu display is turned on and off. If the menu display is off, the LCD drivers will still display data for a grob that is 64 rows high, *regardless* of the actual size of the grob. To see what this looks like, warmstart your HP 48 (hold [ON], press and release [C]), then execute the following secondary:

```
::
  SIXTYFOUR LINECHANGE
  SetDAsTemp
;
```

6.2.8 Display Pointer Examples

To get acquainted with the display grobs, try a quick User-RPL example program that uses SYSEVAL to return the currently displayed grob to the stack and invert the grob. This example uses INVGROB (#122FFh) to invert a grob in level 1 of the stack (the User-RPL command NEG creates a copy of the grob, so INVGROB is easier to use).

```
«
  #12635h SYSEVAL   HARDBUFF returns a pointer to the currently displayed grob
  #122FFh SYSEVAL   INVGROB inverts the grob
  DROP              Drops the pointer (no longer needed)
  7 FREEZE          Postpones display updates
»
```

Inverting the Stack Display. If the program above is executed while the stack display is shown, the stack display will be inverted. A System-RPL equivalent of this program is:

```
::
  HARDBUFF     Returns a pointer to the currently displayed grob
  INVGROB      Inverts the grob
  DROP         Drops the pointer (no longer needed)
  SetDAsTemp   Freeze the display
;
```

Inverting *PICT*. For fun, plot a function, then execute the following program:

```
::
  TOGDISP      Displays PICT
  GBUFF        Returns a pointer to the stack grob
  INVGROB      Inverts the grob
  DROP         Drops the pointer (no longer needed)
  SetDAsTemp   Freeze the display
;
```

Rolling the Menu Display. For more fun, use `LINECHANGE` to scroll the menu out of the display and back in again. This program uses `SLOW` to let you see the menu grob move.

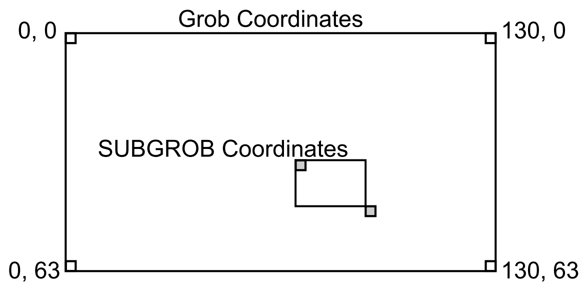
```
SCRMEN 80.5 Bytes Checksum #1B05h
( → )
::
  OLASTOWDOB!                         Clears saved command name
  CKONOLASTWD                         No arguments
  HARDBUFF DUPGROBDIM DROP CHECKHEIGHT  Verify that the display grob is 64 rows high
  SIXTYFOUR FIFTYSIX DO               Loop from 56 to 63
    INDEX@ LINECHANGE SLOW SLOW       Use LINECHANGE to set where menu is displayed
  LOOP
  WaitForKey 2DROP                    Wait for a key, discard keycode and plane
  NINE ONE DO                         Prepare to loop from 63 to 56
    SIXTYTHREE INDEX@ #- LINECHANGE   Use LINECHANGE to set where menu is displayed
    SLOW SLOW
  LOOP
;
```

6.3 Graphics Coordinates

System-RPL objects that work with graphics use internal binary integers to represent pixel coordinates. The upper-left pixel of a grob is always #0,#0.

6.3.1 Subgrob Coordinates

Operations that need to describe the lower-right boundary of an area usually refer to the pixel one row down and one column to the right of the intended area. For example, if `SUBGROB` will be used to create a grob from a larger grob, the coordinates #30 #20 #36 #28 would describe a region beginning on the 31st column and the 21st row in the source grob that is six rows high and eight pixels wide. Other objects that use this convention include `GROB!ZERO` and `GROB!ZERODRP`.



6.3.2 User Pixel Coordinate - Bint Conversion

If you're writing a graphics command that extends the User-RPL command set, you may wish to accept graphics coordinates from the user as a list of two user binary integers like `{ #5d #17d }`. The object `2HXSLIST?` converts this type of list into two bints, ready for use in System-RPL. If the list contains other than two elements that are user binary integers a `Bad Argument Type` error will be generated.

`2HXSLIST?`	#51532h
Converts user pixel coordinates to two bints	
`{ #x #y }` → #x #y	

To return a coordinate to the user as a user binary integer, use the object `#>HXS` (see *Hex String Conversions* on page 96). For example, to return the size of a grob to the user as two user binary integers, use this code:

```
::
  GROBDIM            ( #height #width )
  #>HXS SWAP #>HXS   ( hxs_width hxs_height )
;
```

6.3.3 User-Unit to Pixel Conversion

The following objects use the information in PPAR to convert between user units and pixel coordinates. If PPAR doesn't exist when these are executed, a default PPAR will be created. If you're working on code for plotting, be aware that these routines carry the burden of validating PPAR.

`C%>#`	#4F408h
Converts complex number user-unit coordinates to bint pixel coordinates	
C%(x,y) → #x #y	
`DOC>PX`	#4F179h
Converts complex number user-unit coordinates to user binary integer pixel coordinates	
C%(x,y) → `{ #x #y }`	
`DOPX>C`	#4F0ACh
Converts user binary integer pixel coordinates to complex number user-units	
`{ #x #y }` → C%(x,y)	

6.3.4 Accessing PPAR

The following objects provide access to the user variable PPAR and its contents.

`CHECKPVARS`	#4A9AFh
Validate and return the current contents of PPAR. Issues `Invalid PPAR` error if PPAR is invalid. Creates and returns default PPAR if PPAR is nonexistent.	
→	{ *ppar* }

`GETSCALE`	#4ADB0h
Returns user-unit distance across 10 pixels	
→	%xscale %yscale

`PUTSCALE`	#4AE3Ch
Sets user-unit distance across 10 pixels (does not change center of *PICT*)	
%xscale %yscale →	

Note that each of the following objects carries the burden of validating PPAR.

`GETPMIN&MAX`	#4B0DAh
Returns the current PMIN and PMAX entries from PPAR	
→	C%PMIN C%PMAX

`GETXMIN`	#4B10Ch
Returns the current Xmin coordinate	
→	%Xmin

`GETXMAX`	#4B139h
Returns the current Xmax coordinate	
→	%Xmax

`GETYMIN`	#4B120h
Returns the current Ymin coordinate	
→	%Ymin

`GETYMAX`	#4B14Dh
Returns the current Ymax coordinate	
→	%Ymax

`PUTXMIN`	#4B166h
Stores a new Xmin coordinate	
%Xmin →	

`PUTXMAX`	#4B1ACh
Stores a new Xmax coordinate	
%Xmax →	

`PUTYMIN`	#4B189h
Stores a new Ymin coordinate	
%Ymin →	

`PUTYMAX`	#4B1CFh
Stores a new Ymax coordinate	
%Ymax →	

6.4 Displaying Text*PICT*

The HP 48 has three built-in fonts. Objects are provided that support text display using the medium and large size fonts in fixed display regions. Use of the small font or arbitrary locations in a grob or display grob requires the use of objects like `$>grob`, `GROB!`, and `XYGROBDISP`.

6.4.1 Medium Font Display Objects

The following objects display text in the stack grob using the medium font. Each row is truncated to 22 characters or blank filled. The object `Disp5x7` breaks lines at carriage-returns. Each object displays text beginning at the left edge of ABUFF, *except* for `DISPROW1*` and `DISPROW2*`, which display text relative to the window corner.

Name	Description / Stack	Address
`DISPROW1`	Displays text on row 1 (pixel rows 0-7) $ →	#1245Bh
`DISPROW1*`	Displays text on row 1 relative to the window corner $ →	#12725h
`DISPROW2`	Displays text on row 2 (pixel rows 8–15) $ →	#1246Bh
`DISPROW2*`	Displays text on row 2 relative to the window corner $ →	#12748h
`DISPROW3`	Displays text on row 3 (pixel rows 16–23) $ →	#1247Bh
`DISPROW4`	Displays text on row 4 (pixel rows 24–31) $ →	#1248Bh
`DISPROW5`	Displays text on row 5 (pixel rows 32–39) $ →	#1249Bh
`DISPROW6`	Displays text on row 6 (pixel rows 40–47) $ →	#124ABh
`DISPROW7`	Displays text on row 7 (pixel rows 48–55) $ →	#124BBh
`DISPN`	Displays text on the specified row $ #row →	#12429h
`Disp5x7`	Displays up to #max rows of text starting on the specified row $ #row #max →	#3A4CEh
`DISPSTATUS2`	Displays a string in the first two text rows $ →	#1270Ch

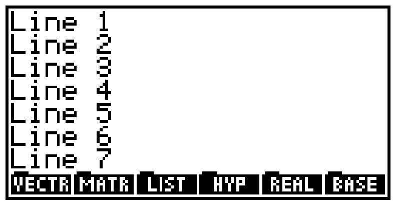
6.4.2 Displaying Temporary Messages

The following objects display a message in the top two lines. The display lines used are preserved and restored.

Name	Description / Stack	Address
`FlashMsg`	Displays a message. $ →	#12B85h
`FlashWarning`	Displays a message and beeps $ →	#38926h

The program MDISPN illustrates the medium font display lines:

```
MDISPN 65.5 Bytes Checksum #56AFh
( → )
::
  CKONOLASTWD OLASTOWDOB!                 Clear saved command name, no arguments
  RECLAIMDISP ClrDA1IsStat                Claim the display, suspend the clock
  EIGHT ONE DO                            Loop for seven lines
    INDEX@ "Line " OVER UNCOERCE DECOMP$ &$   Build the display string
    SWAP DISPN                            Display the string
  LOOP
  SetDAsTemp                              Freeze the display
;
```



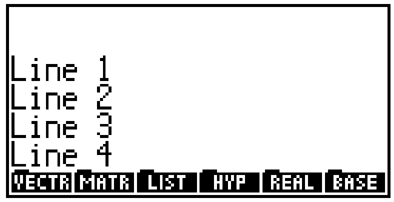
6.4.3 Large Font Display Objects

The following objects display text in the stack grob using the large font. Each row is truncated to 22 characters and blank filled.

`BIGDISPROW1`	#12415h
Displays text on large font row 1 (pixel rows 16–25) $ →	
`BIGDISPROW2`	#12405h
Displays text on large font row 2 (pixel rows 26–35) $ →	
`BIGDISPROW3`	#123F5h
Displays text on large font row 3 (pixel rows 36–45) $ →	
`BIGDISPROW4`	#123E5h
Displays text on large font row 4 (pixel rows 46–55) $ →	
`BIGDISPN`	#123C8h
Displays text on the specified large font row $ #row →	

The program BDISPN illustrates the large font display lines:

```
BDISPN 65.5 Bytes Checksum #875Eh
( → )
::
  CKONOLASTWD OLASTOWDOB!                 Clear saved command name, no arguments
  RECLAIMDISP ClrDA1IsStat                Claim the display, suspend the clock
  FIVE ONE DO                             Loop for four lines
    INDEX@ "Line " OVER UNCOERCE DECOMP$ &$   Build the display string
    SWAP BIGDISPN                         Display the string
  LOOP
  SetDAsTemp                              Freeze the display
;
```



6.5 Basic Grob Tools

The objects described below describe a series of tools for basic grob manipulation.

6.5.1 Creating Grobs

The object `MAKEGROB` is the System-RPL object that does the work for the User-RPL command `BLANK`. The height and width are specified with bints.

`MAKEGROB`			#1158Fh
Creates a blank grob			
#height #width	→	grob	

The following objects create a grob representation of an object.

`$>grob`			#11F80h
Creates a grob from a string using the small font			
$	→	grob	
`$>GROB`			#11D00h
Creates a grob from a string using the medium font			
$	→	grob	
`$>BIGGROB`			#11CF3h
Creates a grob from a string using the large font			
$	→	grob	
`Symb>HBuff`			#659DEh
Creates an EquationWriter representation of an expression in HARDBUFF (may enlarge HARDBUFF)			
'expression'	→		

6.5.2 Finding Grob Dimensions

The following objects return the dimensions of a grob.

`DUPGROBDIM`			#5179Eh
Returns a grob and its dimensions			
grob	→	grob #height #width	
`GBUFFGROBDIM`			#5187Fh
Returns the dimensions of the graphics grob (*PICT*)			
	→	#height #width	
`GROBDIM`			#50578h
Returns the dimensions of a grob			
grob	→	#height #width	
`GROBDIMw`			#63C04h
Returns the width of a grob			
grob	→	#width	

6.5.3 Extracting a Subgrob

The object `SUBGROB` returns a new grob copy of a specified region in a grob. Remember that the lower-right corner is specified by the pixel one row down and one column to the right of the desired region (see *Graphics Coordinates* on page 120).

`SUBGROB`			#1192Fh
Returns a subgrob			
grob $\#x_1$ $\#y_1$ $\#x_2$ $\#y_2$	→	subgrob	

6.5.4 Inverting a Grob

The object `INVGROB` inverts the pixels in a grob.

`INVGROB`			#122FFh
Inverts a grob			
grob	→	grob'	

6.5.5 Combining Graphics Objects

The objects `GROB!` and `GROB+#` place one grob's data within another grob. Note that `GROB!` does no range checking, but `GROB+#` does the work for the User-RPL commands `GOR` and `GXOR`, and so does the same range checking. The object `XYGROBDISP` places a grob in the current display grob (HARDBUFF).

WARNING
Some of these objects *do not* perform any range checking. If you specify a graphics operation that would extend beyond the confines of the grob arguments, you will corrupt memory.

`GROB!`			#11679h
Stores level 4 grob into level 3 grob at specified coordinates			
$grob_{source}$ $grob_{target}$ #x #y	→		
`GROB+#`			#4F78Ch
If *flag* is TRUE, ORs $grob_{source}$ into $grob_{target}$, otherwise XORs grob data			
flag $grob_{target}$ $grob_{source}$ #x #y	→		
`XYGROBDISP`			#128B0h
Places a grob into HARDBUFF, resizing HARDBUFF if needed			
#x #y grob	→		

The object `CKGROBFITS` is useful for ensuring that a grob will fit into another grob when you're going to use `GROB!` and have doubts about the size of the grob being added. `CKGROBFITS` will truncate the grob being added so that a `GROB!` operation will not corrupt memory.

`CKGROBFITS`			#4F7E6h
Ensures that $grob_{new}$ will fit on $grob_{target}$ at the specified coordinates			
$grob_{target}$ $grob_{new}$ #x #y	→	$grob_{target}$ $grob_{new}$' #x #y	

6.5.6 Clearing a Grob Region

The objects `GROB!ZERO` and `GROB!ZERODRP` clear a grob's pixels in a specified region.

`GROB!ZERO`			#11A6Dh
Clears the pixels in the specified region			
grob $\#x_1$ $\#y_1$ $\#x_2$ $\#y_2$	→	grob	
`GROB!ZERODRP`			#6389Eh
Clears the pixels in the specified region and drops the pointer to the grob			
grob $\#x_1$ $\#y_1$ $\#x_2$ $\#y_2$	→		

6.6 Drawing Tools

The following objects are available for drawing lines, setting pixels, etc. Notice that these objects refer either to the stack grob (ABUFF), or the graphics grob (*PICT*). Remember that the upper-left corner of a grob has the coordinates #0 #0 (see *Graphics Coordinates* on page 120).

6.6.1 Line Drawing

Note that line drawing commands require $x_2 \geq x_1$, so you may wish to use `ORDERXY#` to ensure the correct order of parameters.

`ORDERXY#`	Asserts left-to-right order for line-drawing coordinates	$\#x_1\ \#y_1\ \#x_2\ \#y_2 \rightarrow \#x_1\ \#y_1\ \#x_2\ \#y_2$	#51893h
`LINEOFF`	Turns off a line of pixels in the stack display (ABUFF)	$\rightarrow$	#50B08h
`LINEOFF3`	Turns off a line of pixels in the graphics display (GBUFF)	$\#x_1\ \#y_1\ \#x_2\ \#y_2 \rightarrow$	#50ACCh
`LINEON`	Turns on a line of pixels in the stack display (ABUFF)	$\#x_1\ \#y_1\ \#x_2\ \#y_2 \rightarrow$	#50B17h
`LINEON3`	Turns on a line of pixels in the graphics display (GBUFF)	$\#x_1\ \#y_1\ \#x_2\ \#y_2 \rightarrow$	#50AEAh
`TOGLINE`	Toggles a line of pixels in the stack display (ABUFF)	$\#x_1\ \#y_1\ \#x_2\ \#y_2 \rightarrow$	#50AF9h
`TOGLINE3`	Toggles a line of pixels in the graphics display (GBUFF)	$\#x_1\ \#y_1\ \#x_2\ \#y_2 \rightarrow$	#50ADBh

6.6.2 Pixel Control

The following objects clear, set, and test pixels in either the stack or graphics grob.

`PIXOFF`	Turns off a pixel in the stack display (ABUFF)	#x #y $\rightarrow$	#1383Bh
`PIXOFF3`	Turns off a pixel in the graphics display (GBUFF)	#x #y $\rightarrow$	#1380Fh
`PIXON`	Turns on a pixel in the stack display (ABUFF)	#x #y $\rightarrow$	#1384Ah
`PIXON3`	Turns on a pixel in the graphics display (GBUFF)	#x #y $\rightarrow$	#13825h
`PIXON?`	Tests a pixel in the stack display (ABUFF)	#x #y $\rightarrow$ FLAG	#13992h
`PIXON?3`	Tests a pixel in the graphics display (GBUFF)	#x #y $\rightarrow$ FLAG	#13986h

6.7 Menu Grob Utilities

The following objects create menu label grobs (8 pixels high by 21 pixels wide) given a string as input:

`MakeStdLabel`	#3A328h
Creates a standard label	
$ → grob	
`MakeDirLabel`	#3A3ECh
Creates a directory label	
$ → grob	
`MakeBoxLabel`	#3A38Ah
Creates a label with a “mode box” at the right side	
$ → grob	
`MakeInvLabel`	#3A44Eh
Creates an outline box label	
$ → grob	
`Box/StdLabel`	#3EC99h
Creates a label with a “mode box” at the right side if FLAG is TRUE, otherwise create a label without the mode box	
$ FLAG → grob	
`Std/BoxLabel`	#3ED0Ch
Creates a standard menu label if FLAG is TRUE, otherwise creates a label with a “mode box” at the right side	
$ FLAG → grob	

The following objects are used by the menu system to create and display menu label grobs in the dedicated menu grob (HARDBUFF2). The #col parameters for the menu labels are listed in the table below.

Menu Label Column Numbers		
Softkey Number	**Column (hex)**	**Column (decimal)**
1	0	0
2	16	22
3	2C	44
4	42	66
5	58	88
6	6E	110

`Grob>Menu`	#3A297h
Displays an arbitrary 8x21 grob	
#col grob →	
`Id>Menu`	#3A2DDh
Displays a standard or directory label based on the contents of ID	
#col ID →	
`Seco>Menu`	#3A2C9h
Evaluates a secondary that results in a 8x21 grob, then displays the grob	
#col :: ... ; →	
`Str>Menu`	#3A2B5h
Displays a standard menu label	
#col $ →	

6.8 Built-in Grobs

The following objects are built-in:

`SmallCursor`		#66EF1h
3x5 cursor (outline box)		
	→ grob	
`MediumCursor`		#66ECDh
5x7 cursor (outline box)		
	→ grob	
`BigCursor`		#66EA5h
5x9 cursor (outline box)		
	→ grob	
`CURSOR1`		#13D8Ch
5x9 insert cursor		
	→ grob	
`CURSOR2`		#13DB4h
5x9 replace cursor		
	→ grob	
`MARKGROB`		#5055Ah
X symbol		
	→ grob	
`CROSSGROB`		#5053Ch
+ symbol		
	→ grob	

6.9 Graphics Examples

The following examples are designed to showcase a few of the objects described in this chapter. We hope you'll be inspired to experiment with the possibilities. Each of these examples uses ABUFF — the stack display. We encourage you to use ABUFF instead of GBUFF, since *PICT* is considered a user resource like a variable or flag setting.

6.9.1 Drawing a Grid

Some games, like tic-tac-toe and the Minehunt game (built into the HP 48G/GX) need a grid display. This program produces a grid centered in the stack display with a specified number of rows and columns. The size parameter specifies the size of each square (not counting the box boundary lines).

GRID 181 Bytes Checksum #30Ah
(%Size %Rows %Cols →)

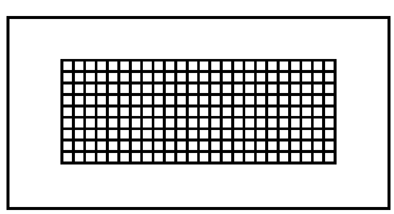
```
::
  0LASTOWDOB! CK3NOLASTWD      Clear saved command name, require three arguments
  CK&DISPATCH1 # 00111         Require three real numbers
  ::
    COERCE2 ROT COERCE #1+     ( #rows #cols #size+1 )
    DUP ROT #* #1+             ( *rows #size+1 #width )
    DUP BINT_131d #>           Verify that the grid is not wider than the display
    case SETSIZEERR            ( #rows #size+1 #width )
    OVER 4ROLL #* #1+          ( #size+1 #width #height )
    DUP SIXTYFOUR #>           Verify that the grid is not taller than the display
    case SETSIZEERR            ( #size+1 #width #height )

    ClrDA1IsStat               Suspend the ticking clock display
    RECLAIMDISP                Assert, clear, and resize ABUFF
    TURNMENUOFF                Turn off the menu display

                               Calculate the addresses of the grid boundaries:
    SIXTYTHREE OVER #-#2/      ( #size+1 #width #height #toprow )
    DUP ROT #+-1               ( #size+1 #width #toprow #botrow )
    BINT_131d 4PICK #-#2/      ( #size+1 #width #toprow #botrow #lfcol )
    DUP 5ROLL #+-1             ( #size+1 #toprow #botrow #lfcol #rtcol )

                               Draw the vertical lines:
    DUP#1+ 3PICK DO            ( #size+1 #toprow #botrow #lfcol #rtcol )
      INDEX@ 5PICK             ( ...  #col #toprow )
      OVER 6PICK               ( ...  #col #toprow #col #botrow )
      LINEON                   ( ...  )
      5PICK                    ( ...  #size+1 )
    +LOOP
                               Draw the horizontal lines:
    3PICK #1+ 5PICK DO         ( #size+1 #toprow #botrow #lfcol #rtcol )
      OVER INDEX@              ( ...  #lfcol #row )
      3PICK OVER               ( ...  #lfcol #row #rtcol #row )
      LINEON                   ( ...  )
      5PICK                    ( ...  #size+1 )
    +LOOP                      ( #size+1 #toprow #botrow #lfcol #rtcol )
    5DROP                      Drop the box parameters
    SetDAsTemp                 Freeze the display
  ;
;
```

The following display was generated with the parameters 3 (size), 9 (rows), and 25 (cols):



For the reader that's interested in assembly language, we suggest you write a code object that replaces the two line drawing loops. For fun, post your code to *comp.sys.hp48* on the Internet. Whose code is fastest?

6.9.2 A Rocket Launch

The `WINDOWXY` and window scrolling objects suggest many possibilities. This program enlarges and scrolls ABUFF to launch a rocket.

```
ROCKET 245.5 Bytes Checksum #E910h
( → )
::
  0LASTOWDOB! CK0NOLASTWD         Clear saved command name, require no arguments
  ClrDA1IsStat RECLAIMDISP        Suspend clock display, assert, clear, and resize ABUFF
                                  Build the "launchpad":
  HARDBUFF2                       Pointer to menu grob
  ZEROZERO 131 EIGHT GROB!ZERO    Clear menu grob
  INVGROB                         Invert menu grob
  ZERO ONE 131 EIGHT GROB!ZERODRP Clear bottom seven rows of menu grob
  ABUFF 55 HEIGHTENGROB           Add 55 rows to the stack display
ASSEMBLE                          Rocket grob
      CON(5)   =DOGROB
      REL(5)   end
      CON(5)   16
      CON(5)   9
      NIBHEX   0100010083008300
      NIBHEX   8300830083008300
      NIBHEX   8300070007000700
      NIBHEX   EF00EF007D103810
end
RPL
  ABUFF 62 40 GROB!               Place rocket in display
  ELEVEN ZERO DO                  Draw the countdown to launch:
  TEN INDEX@ #- UNCOERCE          Real number counts down from 10 to 0
    EDITDECOMP$ $>grob            Convert number to string, then string to grob
    HARDBUFF2                     Pointer to menu grob
    INDEX@                        Get the loop index again
    DUP#0=ITE                     If it's zero ...
      ELEVEN                      ... use 11 for the count x-coordinate base
      :: FIFTEEN VERYSLOW ;       ... otherwise use 15 and delay between numbers
    SWAP TEN #* #+                Calculate x-coordinate for number
    TWO                           Use 2 for y-coordinate
    GROB!                         Put number into menu grob
  LOOP
  56 ONE DO                       Now launch the rocket:
    WINDOWDOWN                    Move the window down one row
    %RAN % .5 %> ?SKIP            There's a 50% chance ...
      :: 67 55 INDEX@ #+ PIXON ;  ... of generating exhaust smoke
    SLOW                          Delay a bit between rows
  LOOP
  RECLAIMDISP                     Resize and clear ABUFF when done
;
```

Chapter 7

Keyboard Utilities

Applications requiring key detection have a variety of options available. In this chapter we illustrate a series of objects and techniques for key detection. These examples use objects described in previous chapters. We first discuss key detection while a program is running, then waiting for a key, and finally some higher-level utilities.

7.1 Key Buffer Utilities

The following objects clear and test the keyboard buffer.

`CHECKKEY`	#04708h
Returns (but does not pop) a pending keycode in the key buffer and TRUE, or FALSE if no key is pending	
→ FALSE	
→ #keycode TRUE	
`FLUSHKEYS`	#00D71h
Clears the key buffer	
→	
`GETTOUCH`	#04714h
Pops a pending keycode from the key buffer and returns TRUE, or returns FALSE if no key is pending	
→ FALSE	
→ #keycode TRUE	
`KEYINBUFFER?`	#42402h
Returns TRUE if any key other than [ON] has been pressed (does not detect the [ON] key)	
→ flag	

Notes:

- The keycodes returned by `CHECKKEY` and `GETTOUCH` do not map directly to key numbers 1 through 49. See *Keycodes* on page 137 below for more information on keycodes.
- These objects don't detect the [ON] key.

7.2 Checking The Keyboard While Running

The HP 48 interrupt system provides a 15-key buffer and a flag that signals that the [ON] key has been pressed. The objects described in this section build upon these basic resources to provide many keyboard detection options.

7.2.1 Detecting the [ON] Key

If a calculation, animation, or simulation process is likely to be either long or infinite, you may wish to let the user signal that the process should stop. The traditional signal is the [ON] key. On the HP 48S/SX models this was referred to as [ATTN] (attention). On the HP 48G/GX this was renamed [CANCEL], but the basic use of the key remained constant. This key is used to interrupt a process, such as an active edit line, a plot in progress, data transfer, or an HP SOLVE calculation. Some processes that work with lists, strings, and matrices also check to see if this key has been pressed.

The interrupt system sets a flag (sometimes called the *attention flag*) when [ON] is pressed. The following objects clear and test this flag.

`ATTNFLGCLR`	#05068h
Clears the attention flag (does not flush the key from the key buffer)	
→	
`ATTN?`	#42262h
Returns TRUE if [ON] has been pressed	
→ flag	

The following program clears the key buffer and attention flag, then begins counting until the object `ATTN?` reports that [ON] has been pressed. The object `FLUSHKEYS` is used to remove the [ON] keystroke from the key buffer.

```
ADDIT 67 Bytes Checksum #DE5h
( → %result )
::
  0LASTOWDOB! CK0NOLASTWD      Clear protection word, no arguments
  ClrDA1IsStat RECLAIMDISP     Turn off clock, clear ABUFF
  TURNMENUOFF                  Turn off the menu
  0%                           Initial value of counter
  ATTNFLGCLR                   Clear the attention flag
  BEGIN
    ATTN? NOT                  Run until [ON] been pressed
  WHILE
    DUP EDITDECOMP$ DISPROW4   Decompile and display counter
    %1+                        Increment counter
  REPEAT
  FLUSHKEYS ATTNFLGCLR         Flush key buffer, clear attention flag
  ClrDAsOK                     Signal display needs to be redrawn
;
```

7.2.2 Detecting Any Key

The object `KEYINBUFFER?` may be used in conjunction with `ATTN?` to detect if any key has been pressed. In practical terms, an application that does this will probably want to use `FLUSHKEYS` and `ATTNFLGCLR` at the end (as shown in the previous example).

KEYINBUFFER? Example: This example is structured much like the ADDIT example, but just uses `KEYINBUFFER?` to look at the whole keyboard.

```
KB 56.5 Bytes Checksum #35EFh
( → %result )
::
  0LASTOWDOB! CK0NOLASTWD       Clear protection word, no arguments
  ClrDA1IsStat RECLAIMDISP      Turn off clock, clear ABUFF
  TURNMENUOFF                   Turn off the menu
  0%                            Initial value of counter
  BEGIN
    KEYINBUFFER? NOT            Has a key been pressed?
  WHILE
    DUP EDITDECOMP$ DISPROW4    Decompile and display counter
    %1+                         Increment counter
  REPEAT
  ClrDAsOK                      Signal display needs to be redrawn
;
```

When you run KB, notice that the [ON] key is not detected, and that the keystroke detected is executed after KB ends. It's also important to notice that the shift keys are treated like any other key in this instance.

SCRIBE Example: This example is more involved than ADDIT and KB, mostly for fun. The object `ATTN?` is used in the same manner as illustrated in ADDIT, but the program also uses `GETTOUCH` to check the rest of the keyboard.

```
SCRIBE 331.5 Bytes Checksum #D363h
( → )
::
  OLASTOWDOB! CKONOLASTWD                 Clear protection word, no arguments
  ClrDA1IsStat RECLAIMDISP                Turn off clock, clear ABUFF
  TURNMENUOFF                             Turn off the menu
  SIXTYFOUR                               Initial X position
  THIRTYTWO                               Initial Y position
  ONE                                     Initial X step Initial Y step
  ONE                                     Running flag
  TRUE
  {
    LAM Xpos LAM Ypos
    LAM Xstep LAM Ystep
    LAM Running                           Bind local variables
  } BIND                                  Clear key buffer and [ATTN] flag
  FLUSHKEYS ATTNFLGCLR
  BEGIN                                   Has a key been pressed?
    GETTOUCH
    ITE                                   Yes, drop keycode and signal FALSE
      DROPFALSE                           No, signal TRUE to keep running
      TRUE                                Has [ATTN] been pressed?
    ATTN? NOT                             AND flags together
    AND                                   If neither even happened, move point:
  WHILE                                   Add step to x position
    LAM Xpos LAM Xstep #+                 If at left edge,
    DUP MINUSONE #= IT                      then reverse direction
      :: #2+ ONE ' LAM Xstep STO ;        If at right edge,
    DUP BINT_131d #= IT                     then reverse direction
      :: #2- MINUSONE ' LAM Xstep STO ;   Save copy on stack for PIXON, store new value
    DUP ' LAM Xpos STO                    Add step to y position
    LAM Ypos LAM Ystep #+                 If at top,
    DUP MINUSONE #= IT                      then reverse direction
      :: #2+ ONE ' LAM Ystep STO ;        If at bottom,
    DUP SIXTYFOUR #= IT                     then reverse direction
      :: #2- MINUSONE ' LAM Ystep STO ;   Save copy on stack for PIXON, store new value
    DUP ' LAM Ypos STO                    Turn on pixel
    PIXON
  REPEAT                                  When done, clear [ATTN] flag
  ATTNFLGCLR                              Signal display needs to be redrawn
  ClrDAsOK
;
```

7.3 Waiting For a Key

While the previous objects are helpful for detecting a key while a program is running, they are not particularly useful if your application is just waiting for the user to press a key. There no sense in running down the batteries!

The object `WaitForKey` does all the hard work for you — returning a fully-formed keystroke specifying the keycode and shift plane. While `WaitForKey` is running, the calculator is placed in a low-power state, conserving batteries.

When `WaitForKey` returns, the keycode and shift plane numbers are returned as bints. The keycode numbering is in row order starting at the top left of the keyboard, running from 1 to 49. The planes are numbered 1 to 6:

Plane	Description
1	Unshifted
2	Left-shifted
3	Right-shifted
4	Alpha
5	Alpha left-shifted
6	Alpha right-shifted

`WaitForKey` #41F65h
Waits in a low power state for a fully-formed keystroke
→ #keycode #plane

The program WKEY displays the keycode and shift plane detected by `WaitForKey` until the [ON] key is pressed. In this example, we use the `BEGIN` ... `UNTIL` loop, just to be different.

WKEY 99.5 Bytes Checksum #B4CAh
(→)

```
::
  OLASTOWDOB! CK0NOLASTWD               Clear protection word, no arguments
  ClrDA1IsStat RECLAIMDISP              Turn off clock, clear ABUFF
  TURNMENUOFF                           Turn off the menu
  BEGIN
    WaitForKey UNCOERCE2                Get keycode and shift plane as real numbers
    "Keycode: " 3PICK EDITDECOMP$ &$ DISPROW3   Display keycode
    "Plane:   " SWAP EDITDECOMP$ &$ DISPROW4    Display shift plane
  UNTIL
  SetDAsTemp                            Freeze the display
;
```

7.4 Keycodes

Unlike the keycodes returned by `WaitForKey`, the keycodes returned by `CHECKKEY` and `GETTOUCH` do not map directly to key numbers from 1 to 49. To see what keycodes are returned, try the program KCODE:

KCODE 64.5 Bytes Checksum #5CFFh
(→)

```
::
  OLASTOWDOB! CK0NOLASTWD          Clear protection word, no arguments
  ClrDA1IsStat RECLAIMDISP         Turn off clock, clear ABUFF
  TURNMENUOFF                      Turn off the menu
  BEGIN
    ATTN? NOT                      Run until [ON] been pressed
  WHILE
    GETTOUCH NOT?SEMI              Loop again if no key in buffer
    UNCOERCE EDITDECOMP$ DISPROW4  Decompile and display keycode
  REPEAT
  FLUSHKEYS ATTNFLGCLR             Flush key buffer, clear attention flag
  ClrDAsOK                         Signal display needs to be redrawn
;
```

As you study KCODE.S, remember that `NOT?SEMI` works here because the compiler places `::` and `;` around the code between `WHILE` and `REPEAT`. To see this, look at the file KCODE.A after KCODE has been compiled. Notice that the [ON] key is not trapped *except* by detecting the attention flag.

The object `CodePl>%rc.p` converts a keycode and plane pair into a real number in RC.P format (as used by user key assignments):

`CodePl>%rc.p`	#41D92h
Converts keycode and plane bints into real number rc.p key address	
#keycode #plane → %rc.p	

The inverse conversion is provided by the object `Ck&DecKeyLoc`:

`Ck&DecKeyLoc`	#41CA2h
Converts real number rc.p key address into keycode and plane bints	
%rc.p → #keycode #plane	

7.5 Repeating Keys

Two objects are available for implementing repeating key procedures. Each takes a keycode and procedure from the runstream and keeps these on the stack. This implies that the object being executed should not alter the stack. In the example fragment below, *`object`* is executed as long as key seventeen is held down:

```
:: ...  REPEATER SEVENTEEN object ...  ;
```

The first object, `REPEATER` has an initial delay of 300 ms, and a 15 ms delay between events. The second, `REPEATERCH`, lacks the long delays, making it well-suited for moving objects around on the screen.

`REPEATER`	#40E88h
Repeats 2nd following object in runstream while the specified key is down	
→	
`REPEATERCH`	#51735h
Repeats 2nd following object in runstream while the specified key is down	
→	

The next example uses `REPEATER` to increment or decrement a number in the display. Try compiling this program with `REPEATER` as shown, then use `REPEATERCH` to see the difference in key response.

```
RPT 172.5 Bytes Checksum #EDD9h
( → )
::
  OLASTOWDOB! CKONOLASTWD                 Clear protection word, no arguments
  ClrDA1IsStat RECLAIMDISP                Turn off clock, clear ABUFF
  TURNMENUOFF                             Turn off the menu
  ' :: 1GETLAM %1+ DUP EDITDECOMP$
    DISPROW4 1PUTLAM ;                    Action for [+] key
  ' :: 1GETLAM %1- DUP EDITDECOMP$
    DISPROW4 1PUTLAM ;                    Action for [-] key
  %0                                      Initial counter value
  ' NULLLAM THREE NDUPN                   Three null temporary variable names
  DOBIND                                  Create the temporary environment
  3GETLAM EVAL                            Increment and display the counter
  BEGIN
    ::
      WaitForKey                          Get keycode and shift plane as real numbers
      DROP                                Ignore the shift plane for this example
      FORTYFOUR #=casedrop                Check for [-]
        ::
          REPEATER FORTYFOUR 2GETEVAL     Subtract once, repeat as long as key is down
          FALSE                           Continue the loop
        ;
      FORTYFIVE #=casedrop TRUE           If [ON] pressed, drop counter and end loop
      FORTYNINE #= case                   Check for [+]
        ::
          REPEATER FORTYNINE :: 3GETLAM EVAL ;  Add once, repeat as long as key is down
          FALSE                           Continue the loop
        ;
      DoBadKey FALSE                      Beep, continue the loop for all other keys
    ;
  UNTIL
  ABND                                    Abandon the temporary environment
  ClrDAsOK                                Signal to redraw the display
;
```

When compiled with REPEATERCH, the size is 172.5 bytes and the checksum is #9561h.

7.6 InputLine

The object InputLine does the work for the user word INPUT. While this interface is not as attractive as an input form (G series only), it's handy for an occasional prompt and parses the input line if you wish.

When executed, InputLine does the following:

- Displays the status area, clears the stack area, and displays a prompt
- Initializes the command line and edit modes
- Displays a menu
- Accepts input from the command line as a string
- Optionally parses, or parses and evaluates the input string
- Returns a flag indicating the way the command line was terminated

`InputLine`	#42F44h
Accepts input from the user, optionally parsing and evaluating the input string	
$Prompt $Input CursorPos #Mode #Entry #Alpha Menu #Row Abort #Action	→ FALSE
	→ $Input TRUE
	→ $Input ob TRUE
	→ ... TRUE

7.6.1 Input Parameters

The ten input parameters are:

$Prompt	A string prompt displayed in display area 2a. This string may contain a newline character.
$Input	The default input string.
CursorPos	The initial cursor position. This can be specified either as a bint character number or a list of two bints specifying the row and column position. Use #0 to specify the end of a row or column.
#Mode	The initial insert/replace mode. Use #0 for the current mode, #1 for insert mode, or #2 for replace mode.
#Entry	The initial entry mode. Use #0 for the current mode + program entry mode, #1 for program/immediate entry, or #2 for program/algebraic entry mode.
#Alpha	The initial alpha-lock mode. Use #0 for the current alpha lock mode, #1 for alpha locked, #2 for alpha unlocked.
Menu	The initial edit menu. This menu specification takes the same form as ParOuterLoop menus, discussed in the next section on page 145.
#Row	The first row of the menu to be displayed (usually specified as #1 for the first menu row).
Abort	A flag specifying the action of the [ON] key when characters are present in the command line. If TRUE, [ON] aborts, returning FALSE. If FALSE, [ON] simply clears the command line.
#Action	Specifies post-command-line processing if terminated by the [ENTER] key. Use #0 to return the input string with no processing, #1 to parse the input string, return the input string and the resulting object, or #2 to parse the input string and evaluate the resulting object. If parsing is required, the command line will not terminate until a valid object is entered.

For a really simple example, consider a prompt for the user's name:

```
:: ... "Name?" NULL$ ZERO ONE ONE ONE NULL{} ONE FALSE ZERO InputLine ... ;
```

This example has a null input string, sets the cursor at the end of the (empty) line, sets program entry mode, locks the alpha mode on, has no menu, specifies that [ON] clears a non-null command line, and does not parse the result.

Input Menu Objects. The menu specification can be as simple or as complicated as you like. Several objects are available that replicate the standard edit menu or components of this menu. The standard edit menu is `EditMenu`:

`EditMenu`	#3BDFAh
The standard command line edit menu	
	→ { menu }

A disadvantage of using `EditMenu` is the presence of the ↑STK menu key (the interactive stack key). If you are writing a closed application, you may have objects on the stack that should not be seen by the user, tampered with, removed, or reordered. To get past this problem, use the individual components that make up `EditMenu` as shown below:

`<SkipKey`	#3E2DDh
The skip-left key	
	→ { key specification }

`>SkipKey`		#3E35Fh
The skip-right key		
	→	{ key specification }

`<DelKey`		#3E3E1h
The delete-left key		
	→	{ key specification }

`>DelKey`		#3E4CAh
The delete-right key		
	→	{ key specification }

`TogInsertKey`		#3E586h
The insert/replace mode key		
	→	{ key specification }

`IStackKey`		#3E5CDh
The interactive stack key		
	→	{ key specification }

To specify a blank key, use `NullMenuKey`:

`NullMenuKey`		#3EC71h
Null menu key		
	→	{ key specification }

For example, a menu that provides the basic edit capabilities but not the interactive stack might look like this:

```
{ <SkipKey >SkipKey <DelKey >DelKey NullMenuKey TogInsertKey }
```

Note that in this example `NullMenuKey` is used as a placeholder. `NullMenuKey` is not needed if used after the last defined key — the system will place a blank keys in the remaining positions for you. A menu with only two edit keys defined in positions two and three and a string in the fifth position would be specified as follows:

```
{ NullMenuKey <DelKey >DelKey NullMenuKey "Jim" }
```

If a string is provided as a menu key object, the menu key label is built from that string, and the string is echoed into the command line at the current cursor position when the menu key is pressed.

7.6.2 InputLine Results

Since `InputLine` accepts a variety of input conditions, the results vary depending on input conditions and user actions. The flag in level one indicates FALSE if the user aborted the command line by pressing [ON]. If this flag is TRUE, the results above level one depend on the #Action parameter. If #Action was #0 or #1, you know there will be one or two objects on the stack. If #Action was #2, you have *no way* of knowing what's on the stack. Most applications that use `InputLine` avoid this case, since there are simply too many ways for the user to enter a procedure that challenges the programmer's assumptions about the state of the machine.

7.6.3 InputLine Examples

The first example, INP1, illustrates a simple prompt for a name. The menu is specified using individual `EditMenu` components and a string to illustrate a simple string-echo key.

INP1 97.5 Bytes Checksum #9FC5h

(→ $ 1 *or* 0)

```
::
  0LASTOWDOB! CKONOLASTWD        Clear protection word, no arguments
  "Enter your name:"             Prompt
  NULL$                          Initial input line
  ZERO                           Initial cursor position
  ONE                            Insert mode
  ONE                            Program/immediate entry mode
  ONE                            Alpha locked
  {                              Menu specification
    <SkipKey
    >SkipKey
    <DelKey
    >DelKey
    TogInsertKey
    "Jim"
  }
  ONE                            Menu row one
  FALSE                          [ON] clears the command line
  ZERO                           No post-command-line processing
  InputLine                      Run the command line
  ITE %1 %0                      Convert the result flag to a real 0 or 1
  ClrDAsOK                       Signal to redraw the display
;
```

The second example, INP2, prompts for a real number, ending only if the user aborts by pressing [ON]. Since `InputLine` doesn't accept a specification for what type of object should be returned, the type check must occur after `InputLine`. To implement this, a loop is used to continue prompting until a real number is entered or the user aborts the command line.

```
INP2 149.5 Bytes Checksum #5EF3h
( → % %1 or %0 )
::
  0LASTOWDOB! CK0NOLASTWD             Clear protection word, no arguments
  BEGIN                               Beginning of type checking loop
  ::
    "Enter a number:"                 Prompt
    NULL$                             Initial input line
    ZERO                              Initial cursor position
    ONE                               Insert mode
    ONE                               Program/immediate entry mode
    TWO                               Alpha off
    {                                 Menu specification
      <SkipKey
      >SkipKey
      <DelKey
      >DelKey
      TogInsertKey
    }
    ONE                               Menu row one
    FALSE                             [ON] clears the command line
    ONE                               Parse command line, require a valid object
    InputLine                         Run the command line
    NOTcase :: %0 TRUE ;              End loop, return %0 if user aborted with [ON]
    DUPTYPEREAL?                      Is the object a real number?
    case
      ::                              If so,
        SWAPDROP                        Discard the input string
        %1                              Return %1 to signal a real number result
        TRUE                            Signal the end of the loop
      ;
    2DROP                             If not, discard object and input string
    "Real Number Only" FlashWarning     Display a warning
    FALSE                               and signal the loop needs to continue
  ;
  UNTIL                               End of type checking loop
  ClrDAsOK                            Signal to redraw the display
;
```

The third example, INP3, expands the INP2 example with a |HELP| menu key. A different method for displaying a message is used. The help and warning messages are the same, but you could expand the example to use different messages. The techniques used for the HELP key are described in further detail in the next section.

INP3 405 Bytes Checksum #47C9h
(→ % %1 *or* %0)

```
::
  0LASTOWDOB! CK0NOLASTWD                           Clear protection word, no arguments
  ' ::                                              Subroutine to show message
      ABUFF TEN THIRTY 121 FORTYONE SUBGROB           Save display area on stack
      ABUFF TEN THIRTY 121 FORTYONE GROB!ZERODRP      Clear message area
      TEN THIRTY 121 THIRTY LINEON                    Draw box
      121 THIRTY 121 FORTY LINEON
      TEN FORTY 121 FORTY LINEON
      TEN THIRTY TEN FORTY LINEON
      "ENTER A REAL NUMBER" $>grob                    Create message grob
      ABUFF TWENTYFIVE THIRTYTHREE GROB!              Put message in display
      VERYSLOW VERYSLOW                               Wait 600 ms
    ;
  ' :: ABUFF TEN THIRTY GROB! ;                     Subroutine to restore display
  ' LAM ShowHelp
  ' LAM HelpOff
  TWO DOBIND                                        Create temporary environment
  BEGIN
  ::
    "Enter a number:"                               Prompt
    NULL$                                           Initial input line
    ZERO                                            Initial cursor position
    ONE                                             Insert mode
    ONE                                             Program/immediate entry mode
    TWO                                             Alpha off
    {                                               Menu specification
      <SkipKey >SkipKey
      <DelKey >DelKey
      TogInsertKey
      {                                             Sixth menu key specification:
        "HELP"                                        Label
        ::
          TakeOver                                    Signal takeover secondary
          LAM ShowHelp EVAL                           Display message, wait 600 ms
          REPEATER SIX NOP                            Do nothing while 6th softkey is down
          LAM HelpOff EVAL                            Restore display
        ;
      }
    }
    ONE                                             Menu row one
    FALSE                                           [ON] clears the command line
    ONE                                             Parse command line, require valid obj
    InputLine                                       Run the command line
    NOTcase :: %0 TRUE ;                            End loop, return %0 if cancelled
    DUPTYPEREAL?                                    Is the object a real number?
    case :: SWAPDROP %1 TRUE ;                      Yes, discard input string, signal done
    2DROP                                           No, discard string and ob,
    LAM ShowHelp EVAL LAM HelpOff EVAL              display message,
    FALSE                                           and signal the loop needs to continue
  ;
  UNTIL                                             End of type checking loop
  ABND                                              Abandon temporary environment
  ClrDAsOK                                          Signal to redraw the display
;
```

Chapter 8

The Parameterized Outer Loop

Applications wishing to take complete control of the keyboard and display can use any of the techniques described so far, but the Parameterized Outer Loop (also known as the POL) provides a flexible, easy-to-use environment. While somewhat daunting to learn at first, the POL should quickly become a trusty part of your toolkit. Since there are many potentially complex relationships between the various components of an application that uses a POL, you may end up reading through the descriptions and examples several times before it all makes sense.

At the simplest level, the Parameterized Outer Loop refreshes the display, accepts and processes keys that you decide are valid and continues until an exit condition is met. The POL is therefore an engine which you may call with parameters specifying its behavior. POL's may be nested to the limits of available memory. In this chapter we'll explore the POL with a series of examples, each doing a little more work than the last one. Since there's a wide variety of ways to use the POL or its components, you'll find yourself mixing and matching techniques presented in these examples.

8.1 Introducing ParOuterLoop Parameters

The POL requires nine parameters and does not return anything. Each key may, of course, place an object on the stack, so the results are non-deterministic *unless* you count objects removed from or placed onto the stack. We begin with a general description of the parameters and an example, then discuss some parameters in greater detail.

`ParOuterLoop`	#38985h
The Parameterized Outer Loop	
Display_ob Hardkey_ob NonAppKey_flag DoStdKeys_flag Softkey_menu #Menurow Suspend_flag Exit_ob Error_ob →	

❾ Display Object The display object is evaluated before each key is evaluated. In the simplest case (where each key performs all display updates), this object is responsible for making sure the current menu is displayed. The first example does just this.

❽ Hardkey Handler The hardkey processing object. This object is first to have a chance at processing each keystroke. This object is described in detail in *Hardkey Handlers* on page 152 below.

❼ NonAppKey Flag A flag which, if FALSE, prevents the standard behavior of keys not defined by the hardkey handler. If this flag is TRUE, then a key not defined by the hardkey handler would execute as specified by the DoStdKeys flag (described next). Note that softkeys are considered "standard keys", and their actions are usually bundled with the softkey definition, so this flag must be TRUE to let the softkey code execute.

❻ DoStdKeys Flag A flag which, if FALSE, allows user key assignments to be processed for keys not defined by the hardkey handler. If TRUE, this flag causes user key assignments to be ignored. It's a good practice to leave this flag TRUE unless you're expecting arbitrary input.

❺ Softkey Menu A list of softkey definitions. These are described in detail in *Softkey Definitions* on page 157. If your application has softkey definitions, NonAppKeyFlag must be TRUE to enable your softkeys.

❹ #Menu Row A binary integer indicating which page of a multiple-page softkey definition should be displayed first. This value is typically ONE.

❸ Suspend Flag If an application will permit the evaluation of arbitrary objects and commands, the system becomes quite vulnerable when the user commands `HALT` or `PROMPT` are executed. In this state, the user has access to the entire system, notably the stack and variable memory. To prevent this, the Suspend flag should always be FALSE, which makes commands like `HALT` & `PROMPT` generate a `Halt Not Allowed` error.

❷ Exit Object The POL evaluates this object after each keystroke, and exits when TRUE is returned.

❶ Error Object This object is evaluated when an error occurs during execution of a key definition. The object can be specified as `' ERRJMP` in the simplest case. If you wish to trap specific errors, this object can be as complex as you like.

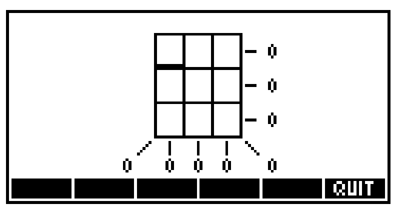
Example: The program POLL displays a number, then enables the [+] and [-] keys to increment and decrement this number. The [OFF] key is enabled, and the softkey `|QUIT|` is used to provide the exit signal. In the listing below, the nine `ParOuterLoop` parameters are highlighted with the numbers ❶ through ❾ indicating each parameter's stack level.

```
POL1 330.5 Bytes Checksum #CA87h
( → )
DEFINE kpNoShift ONE
DEFINE kpRightShift THREE
DEFINE kcRightShift FORTY
DEFINE kcMinus FORTYFOUR
DEFINE kcOn FORTYFIVE
DEFINE kcPlus FORTYNINE

::
    OLASTOWDOB! CKONOLASTWD                          Clear saved command name, no
                                                     arguments
    ClrDA1IsStat RECLAIMDISP                         Suspend clock, clear display
    FALSE                                            Exit flag
    % 1                                              Initial counter value
    ' LAM Running
    ' LAM Value                                      Create temporary environment
    TWO DOBIND                                       Display action
❾   ' ::                                             Display menu if not done already
      DA3OK? ?SKIP :: DispMenu.1 SetDA3Valid ;       Display the counter value
      LAM Value EDITDECOMP$ DISPROW4
    ;
❽   ' ::                                             Hard key handler:
      kpNoShift #=casedrop                           Process primary key plane:
        ::
          DUP#<7 casedrpfls                          Enable soft keys
          kcMinus ?CaseKeyDef                        Process [-] key
            :: TakeOver LAM Value %1- ' LAM Value STO ;
          kcPlus ?CaseKeyDef                         Process [+] key
            :: TakeOver LAM Value %1+ ' LAM Value STO ;
          kcRightShift #=casedrpfls                  Enable right shift key
          DROP 'DoBadKeyT                            Reject all other keys
        ;
      kpRightShift #=casedrop                        Process right shift plane:
        ::
```

```
          kcRightShift #=casedrpfls                  Enable right shift key
          kcOn #=casedrpfls                          Enable [OFF]
          DROP 'DoBadKeyT                            Reject all other keys
        ;
      2DROP 'DoBadKeyT                               Reject all other planes
    ;
❼   TRUE                                             Enable softkeys
❻   TRUE                                             Reject user key definitions
❺   {                                                Softkey menu:
      NullMenuKey                                    Blank menu key 1
      NullMenuKey                                    Blank menu key 2
      NullMenuKey                                    Blank menu key 3
      NullMenuKey                                    Blank menu key 4
      NullMenuKey                                    Blank menu key 5
      {                                              |QUIT| key (6):
        "QUIT"                                       Label text
        :: TakeOver TRUE ' LAM Running STO ;         Key action
      }
    }
❹   ONE                                              Display 1st menu row
❸   FALSE                                            Don't allow HALT or PROMPT
❷   ' LAM Running                                    Exit object
❶   ' ERRJMP                                         Error handler
    ParOuterLoop                                     Run the POL
    ABND                                             Discard temporary environment
    ClrDAsOK                                         Signal to redraw the display
;
```

Example: The program MAGIC implements a magic square puzzle. Use the arrow keys and digit keys to place the digits in a 3x3 grid so that all the rows, columns, and diagonals add up to 15. In the listing below, the nine `ParOuterLoop` parameters are highlighted with the numbers ❶ through ❾ indicating each parameter's stack level.



MAGIC 1488.5 Bytes Checksum #8226h

(→)

```
DEFINE kpNoShift     ONE
DEFINE kpLeftShift   TWO
DEFINE kpRightShift  THREE
DEFINE kcUpArrow     ELEVEN
DEFINE kcLeftArrow   SIXTEEN
DEFINE kcDownArrow   SEVENTEEN
DEFINE kcRightArrow  EIGHTEEN
DEFINE kc7           THIRTYONE
DEFINE kc8           THIRTYTWO
DEFINE kc9           THIRTYTHREE
DEFINE kc4           THIRTYSIX
DEFINE kc5           THIRTYSEVEN
DEFINE kc6           THIRTYEIGHT
DEFINE kcRightShift  FORTY
DEFINE kc1           FORTYONE
```

```
DEFINE kc2           FORTYTWO
DEFINE kc3           FORTYTHREE
DEFINE kc0           FORTYSIX
DEFINE kcOn          FORTYFIVE

DEFINE Row           'L1
DEFINE Col           'L2
DEFINE Running       'L3
DEFINE Data          'L4
DEFINE Highlight     'L5
DEFINE PutDigit      'L6
DEFINE ShowDigit     'L7
DEFINE PutSum        'L8

::
0LASTOWDOB! CK0NOLASTWD                      Clear saved cmd name, no arguments
ClrDA1IsStat RECLAIMDISP                     Suspend the clock, clear the display

Draw the grid

FOUR ZERO_DO (DO)
  FIFTY INDEX@ TEN #* #+ SIX OVER FORTYTWO LINEON
  FIFTY SIX INDEX@ TWELVE #* #+ EIGHTY OVER LINEON
LOOP

THREE ZERO_DO (DO)
82 TWELVE INDEX@ TWELVE #* #+ 85 OVER LINEON
FIFTYFIVE INDEX@ TEN #* #+ FORTYFOUR OVER FORTYEIGHT LINEON
LOOP

FORTYFOUR FORTYEIGHT FORTYEIGHT FORTYFOUR LINEON
82 FORTYFOUR 86 FORTYEIGHT LINEON

Create temporary variables

ONEONE                                       Default X and Y grid location
FALSE                                        Exit flag
{ ZERO ZERO ZERO ZERO ZERO ZERO ZERO ZERO ZERO }   Cache of grid bints
TOTEMPOB

' :: ( Highlight ) ( → )                     Subroutine to draw underscore
  FORTYONE LAM Col TEN #* #+                 Calculate X coordinate of line start
  FIVE LAM Row TWELVE #* #+                  Calculate Y coordinate of line start
  OVER EIGHT #+ OVER                         Line end coordinates
  TOGLINE                                    Draw a toggled pixel line
;

' :: ( PutDigit ) ( #digit → )               Subroutine to store digit in cache
  LAM Row #1- THREE #* LAM Col #+            Calculate digit position in cache
  LAM Data 3PICK
  EQUALPOSCOMP                               Is digit already stored?
  DUP#0= ITE
    :: DROP LAM Data ;                       No, prepare to store digit
    ::
      ZEROSWAP LAM Data                      Yes, store 0 in old position
      LAM ShowDigit EVAL PUTLIST
    ;
```

```
  LAM ShowDigit EVAL                                  Display digit in grid
  PUTLIST                                             Store new digit in cache
  ' LAM Data STO                                      Re-store the cache
;

' :: ( ShowDigit ) ( #digit #pos {data} → )           Subroutine to display digit
  "\35\3F\49\35\3F\49\35\3F\49" 3PICK SUB$1#          Get X position of digit
  "\09\09\09\15\15\15\21\21\21" 4PICK SUB$1#          Get Y position of digit
  5PICK DUP#0= ITE                                    Is this digit zero?
    :: DROP SPACE$ ;                                    Yes, display a space
    :: FORTYEIGHT #+ #>CHR CHR>$ ;                      No, display the digit
  $>GROB XYGROBDISP                                   Convert to grob and put in display
;

' :: ( PutSum ) ( #x #y Pos1 Pos2 Pos3 → #sum )       Subroutine to calc and display sum
  LAM Data DUPDUP                                     Get three copies of the cache
  4ROLL NTHCOMPDROP                                   Get first digit
  SWAP 4ROLL NTHCOMPDROP                              Get second digit
  ROT 4ROLL NTHCOMPDROP                               Get third digit
  #+ #+ DUP 4UNROLL                                   Calculate sum and save copy
  DUP UNCOERCE EDITDECOMP$                            Decompile digit
  $>grob SWAP                                         Make digit into grob
  TEN #< IT                                           If sum is less than 10
    :: SIX EIGHT MAKEGROB DUPUNROT TWO ZERO GROB! ;     then enclose in two-digit-wide grob
  XYGROBDISP                                          Display sum grob
;                                                     Store new digit in cache

{
  LAM Row
  LAM Col
  LAM Running
  LAM Data
  LAM Highlight
  LAM PutDigit
  LAM ShowDigit
  LAM PutSum
}
BIND
```

Put the parameters for the ParOuterLoop on the stack

```
❾ ' ::                                                Display Action
  DA3OK? ?SKIP :: DispMenu.1 SetDA3Valid ;            Display the menu if needed
  LAM Highlight EVAL                                  Turn on the underscore
  ZERO TWENTYONE 88 TEN ONE TWO THREE LAM PutSum EVAL Calculate and display sums
  88 TWENTYTWO FOUR FIVE SIX LAM PutSum EVAL
  88 THIRTYFOUR SEVEN EIGHT NINE LAM PutSum EVAL
  THIRTYSEVEN FIFTY THREE FIVE SEVEN LAM PutSum EVAL
  FIFTYTWO FIFTY ONE FOUR SEVEN LAM PutSum EVAL
  SIXTYTWO FIFTY TWO FIVE EIGHT LAM PutSum EVAL
  72 FIFTY THREE SIX NINE LAM PutSum EVAL
  88 FIFTY ONE FIVE NINE LAM PutSum EVAL
  TRUE EIGHT ZERO_DO (DO)                             Loop to see if all sums were 15
    SWAP FIFTEEN #= AND
  LOOP
  ITE "GOT IT!" "       "                             Decide which string to display
  $>GROB XYGROBDISP                                   Display string
```

```
;

❽ ' ::                                                  Hardkey Handler
  LAM Highlight EVAL                                    Turn off the underscore
  kpNoShift #=casedrop                                  Primary key plane
  ::
    DUP#<7 casedrpfls ( Enable soft keys )
    kcUpArrow ?CaseKeyDef
                                                        [▲]
      :: TakeOver LAM Row DUP#1= casedrop
         DoBadKey #1- ' LAM Row STO ;
    kcDownArrow ?CaseKeyDef
                                                        [▼]
      :: TakeOver LAM Row DUP #3= casedrop
         DoBadKey #1+ ' LAM Row STO ;
    kcLeftArrow ?CaseKeyDef
                                                        [◄]
      :: TakeOver LAM Col DUP#1= casedrop
         DoBadKey #1- ' LAM Col STO ;
    kcRightArrow ?CaseKeyDef
                                                        [►]
      ::
        TakeOver
        LAM Col DUP #3= ITE                             Enable wrap to next row
          :: DROPONE LAM Row DUP #3= ITE
             DROPONE #1+ ' LAM Row STO ;
          #1+
        ' LAM Col STO
      ;

    kc0 ?CaseKeyDef :: TakeOver ZERO LAM PutDigit EVAL ;      [0]
    kc1 ?CaseKeyDef :: TakeOver ONE LAM PutDigit EVAL ;       [1]
    kc2 ?CaseKeyDef :: TakeOver TWO LAM PutDigit EVAL ;       [2]
    kc3 ?CaseKeyDef :: TakeOver THREE LAM PutDigit EVAL ;     [3]
    kc4 ?CaseKeyDef :: TakeOver FOUR LAM PutDigit EVAL ;      [4]
    kc5 ?CaseKeyDef :: TakeOver FIVE LAM PutDigit EVAL ;      [5]
    kc6 ?CaseKeyDef :: TakeOver SIX LAM PutDigit EVAL ;       [6]
    kc7 ?CaseKeyDef :: TakeOver SEVEN LAM PutDigit EVAL ;     [7]
    kc8 ?CaseKeyDef :: TakeOver EIGHT LAM PutDigit EVAL ;     [8]
    kc9 ?CaseKeyDef :: TakeOver NINE LAM PutDigit EVAL ;      [9]
    kcOn ?CaseKeyDef :: TakeOver TRUE ' LAM Running STO ;     [ON] ends the program
    kcRightShift #=casedrpfls                                 [↱]
    DROP 'DoBadKeyT                                           Reject other non-shifted keys
  ;

  kpRightShift #=casedrop                               Right-shift key plane
  ::
    kcRightShift #=casedrpfls                           Enable [↱]
    kcOn #=casedrpfls                                   Enable [OFF]
    DROP 'DoBadKeyT                                     Reject other right-shifted keys
  ;
```

```
  2DROP 'DoBadKeyT                                  Reject other planes
;

❼ ❻ TrueTrue                                        Key control flags

❺ {                                                  Softkey menu
  NullMenuKey
  NullMenuKey
  NullMenuKey
  NullMenuKey
  NullMenuKey
  {
    "QUIT"
    :: TakeOver TRUE ' LAM Running STO ;
  }
}

❹ ❸ ONEFALSE                                        1st row, no suspend
❷ ' LAM Running                                      Exit condition
❶ ' ERRJMP                                           Error handler
ParOuterLoop                                         Run the ParOuterLoop
ABND                                                 Abandon temp environment
ClrDAsOK                                             Signal to redraw the display
;
```

8.2 Temporary Environments and the POL

The object `ParOuterLoop` creates a temporary environment that saves the previous menu system, key handlers, display objects, and so on. This is the mechanism that lets you nest POLs. Unless you're using the individual POL utilities (described later), it's advisable to use named temporary variables as shown in the previous example.

8.3 The Exit Object

The exit object's activity can be as simple as recalling a variables contents or as complex as you like. In the previous example a temporary variable name was supplied as the exit object. If you're writing an application such as an editor, the exit action might make sure the user has "saved information" before permitting an exit.

8.4 The Error Object

The error object gives you a chance to intercept errors that would otherwise terminate your application. In many cases, applications use error traps within key operations to trap anticipated errors, and just provide `ERRJMP` as the error object. Consider a plotting application — an error trap around the calculation for each point would trap math errors, such as divide-by-zero, while a general system error like low memory might be passed out of the POL, terminating the application.

The error object also gives you a chance to try to save information that's in temporary memory. For instance, if your application is an editor, you might want to try to save information in a user variable before the application terminates.

8.5 Display Objects

Display updates can be performed either by a key definition or by the POL display object. The display object is evaluated before each keystroke. The display object has two main responsibilities — display the softkey menu (if needed), and perform display updates not handled by key definitions. The example on the previous page illustrates these two activities. Unless your application doesn't use a menu, the first component is usually present:

```
::
  DA3OK? ?SKIP :: DispMenu.1 SetDA3Valid ;   Display the menu if needed
  ...                                        Perform general display updates
;
```

The DA3 display flag is used to track the status of the menu display. If one of your key definitions changes the menu definition or conditions that would affect the menu display, then executing `ClrDA3OK` would cause the menu to be redisplayed the next time the display object is executed. This is useful for dynamic key labels, which will be illustrated in *Softkey Definitions* on page 157 below.

If no display action is needed other than for the menu, the display object can be coded as follows:

```
::
  DA3OK? ?SEMI              Exit if the menu display is valid
  DispMenu.1 SetDA3Valid    Otherwise display the menu
;
```

If your application has no menu and doesn't need a general display object at all, specify `'NOP`.

8.6 Hardkey Handlers

Every keystroke (including shift modifiers) is processed by the hard key handler. This key handler accepts a key specification in the form of two binary integer codes — a *keycode number* and a *shift plane number*. The handler returns either an object to evaluate and the flag TRUE *or* FALSE to pass the key on the rest of the system.

#keycode #plane	→	object TRUE	*Application defines the key*
#keycode #plane	→	FALSE	*Application does not define the key*

8.6.1 Key and Plane Codes

The previous example, POL1, used DEFINEs for the RPL compiler to make the code easier to read. The file KEYDEFS.H supplied with the HP tools contains definitions for all shift planes and keycodes. To use these definitions in your source code, just add `INCLUDE KEYDEFS.H` to include the definitions.

HP 48 keys are numbered from 1 to 49 in row order starting at the upper left of the keyboard. The shift planes are numbered 1 to 6. Their codes and definitions in KEYDEFS.H are listed below:

Shift Planes					
#plane	*definition*	**Primary Planes**	*#plane*	*definition*	**Alpha Planes**
1	`kpNoShift`	Unshifted	4	`kpANoShift`	Alpha
2	`kpLeftShift`	Left-shifted	5	`kpALeftShift`	Alpha left-shifted
3	`kpRightShift`	Right-shifted	6	`kpARightShft`	Alpha right-shifted

The keycode numbers and definitions in KEYDEFS.H are listed below:

1 kcMenuKey	2 kcMenuKey2	3 kcMenuKey3	4 kcMenuKey4	5 kcMenuKey5	6 kcMenuKey6
7 kcMathMenu	8 kcPrgmMenu	9 kcCustomMenu	10 kcVarsMenu	11 kcUpArrow	12 kcNextRow
13 kcTick	14 kcSto	15 kcEval	16 kcLeftArrow	17 kcDownArrow	18 kcRightArrow
19 kcSin	20 kcCos	21 kcTan	22 kcSqrt	23 kcPower	24 kcInverse
25 kcEnter		26 kcNegate	27 kcEnterExp	28 kcDelete	29 kcBackspace

30 kcAlpha	31 kc7	32 kc8	33 kc9	34 kcDivide
35 kcLeftShift	36 kc4	37 kc5	38 kc6	39 kcTimes
40 kcRightShift	41 kc1	42 kc2	43 kc3	44 kc4
45 kcOn	46 kc0	47 kcPeriod	48 kcSpace	49 kcPlus

8.6.2 Hardkey Handler Structure

Hardkey handlers are typically structured as follows:

```
::
  Unshifted plane?
    Yes, process #keycode for the unshifted plane
  Left-shifted plane?
    Yes, process #keycode for the left-shifted plane
  Right-shifted plane?
    Yes, process #keycode for the right-shifted plane
  Alpha plane?
    Yes, process #keycode for the alpha plane
  Alpha left-shifted plane?
    Yes, process #keycode for the alpha left-shifted plane
  Process #keycode for the alpha right-shifted plane
;
```

Selecting the Key Plane. The object `#=casedrop` (which should have been named `OVER#=casedrop`) is quite useful for key handlers:

`#=casedrop` #618D3h

If #x = #y, drops #x and #y from the stack, executes $\mathrm{object}_{\mathrm{TRUE}}$, and skips the remainder of the secondary, otherwise drops #y, skips $\mathrm{object}_{\mathrm{TRUE}}$, and executes the remainder of the secondary

#x #y → *(#x = #y)*

#x #y → #x *(#x ≠ #y)*

`:: ... #=casedrop objectTRUE ... ;`

Using this object, the key handler begins to take shape:

```
::
  kpNoShift #=casedrop              :: process unshifted keycodes ;
  kpLeftShift #=casedrop            :: process left-shifted keycodes ;
  kpRightShift #=casedrop           :: process right-shifted keycodes ;
  kpANoShiftShift #=casedrop        :: process alpha unshifted keycodes ;
  kpALeftShift #= case              :: process alpha left-shifted keycodes ;
  process alpha right-shifted keycodes
;
```

A key handler that only needs to process two planes, like the POL1 example, would have the following structure:

```
::
  kpNoShift #=casedrop      :: process unshifted keycodes ;
  kpRightShift #=casedrop   :: process right-shifted keycodes ;
  2DROP 'DoBadKeyT          (Reject all other planes)
;
```

or:

```
::
  kpNoShift #=casedrop                  :: process unshifted keycodes ;
  kpRightShift #<> casedrop 'DoBadKeyT  (Reject all other planes)
  process right-shifted keycodes
;
```

The object `'DoBadKeyT` used above generates the invalid key beep, and is described below under *Signaling Invalid Keys* on page 156. Once the plane has been identified, each secondary that processes keycodes now has the following stack diagram:

#keycode	→	object TRUE	*Application defines the key*
#keycode	→	FALSE	*Application does not define the key*

Enabling Specific Standard Keys. Every keystroke, *including* modifier keys, must be handled by the hardkey handler. This means that every plane handler must enable the modifier keys for other allowed planes. Other functions, like [NXT] and [OFF] may be enabled using the same technique. The object `#=casedropfls` (which should have been named `OVER#=casedropfls`) is quite useful here:

`#=casedrpfls` #63547h

If #x = #y, drops #x and #y from the stack, leaves FALSE on the stack and skips the remainder of the secondary, otherwise drops #y and executes the remainder of the secondary.

#x #y	→	FALSE	*(#x = #y)*
#x #y	→	#x	*(#x ≠ #y)*

`:: ... #=casedropfls ... ;`

All well-mannered applications should enable [OFF], since the user might be interrupted at any time. Expanding the example of a hardkey handler that processes only the primary and right-shifted planes from the previous page, the handler now looks like this:

```
::
  kpNoShift #=casedrop
    ::
      kcRightShift #=casedrpfls              Enables
      process remaining unshifted keycodes
    ;
  kpRightShift #=casedrop
    ::
      kcRightShift #=casedrpfls              Enables
      kcOn #=casedrpfls                      Enables [OFF]
      process remaining right-shifted keycodes
    ;
  2DROP 'DoBadKeyT                           Reject all other planes
;
```

Note that the right-shift key is enabled in *both* the primary and right-shifted planes. This lets the user press , then go back to the primary plane by pressing again.

Multi-Page Menus. If your menu has more than six softkeys, you can enable the standard [NXT] key functions using the same technique used for the shift keys. In the primary, left, and right plane handlers, include the line:

`kcNextRow #=casedrpfls`

This enables the following functions:

Keystroke	Purpose
[NXT]	Display the next 6 softkeys
⮌ [NXT]	Display the previous 6 softkeys
⮎ [NXT]	Display the first 6 softkeys

Enabling Softkeys. In the usual case, softkey actions are included as part of each softkey definition. In this situation, softkey actions are initiated by the system *after* the hardkey handler, so the `NonAppKey` flag *must* be TRUE and the hardkey handler must return FALSE for each menu key. Expanding the example on the previous page, the hardkey handler now looks like this:

```
::
  kpNoShift #=casedrop
    ::
      DUP#<7 casedrpfls                 Enables primary softkeys
      kcRightShift #=casedrpfls         Enables ⮎
      kcNextRow #=casedrpfls            Enables [NXT]
      process remaining unshifted keycodes
    ;
  kpRightShift #=casedrop
    ::
      kcRightShift #=casedrpfls         Enables ⮎
      kcOn #=casedrpfls                 Enables [OFF]
      process remaining right-shifted keycodes
    ;
  2DROP 'DoBadKeyT                      Reject all other planes
;
```

Note that only the primary softkey plane is enabled here. Applications like the solver that use left- and right-shifted menu keys *must* include the test for each enabled plane.

Key Definitions. Once you've coded the plane handlers, enabled the modifiers, [OFF], [NXT], and softkeys, you're ready to include the code that is specific to your application. A useful object for coding key handlers is `?CaseKeyDef`:

`?CaseKeyDef`	#3FF1Bh
If #x = #y, drops #x and #y from the stack, leaves the next object in the secondary on the stack and TRUE and skips the remainder of the secondary, otherwise drops #y and executes the remainder of the secondary.	
#x #y → KeyOb TRUE	*(#x = #y)*
#x #y → #x	*(#x ≠ #y)*
`:: ... ?CaseKeyDef` *KeyOb* `... ;`	

Custom key definitions *must* include the object `TakeOver` at the start of the definition to signal a custom definition. This object serves only as a placeholder, and does nothing.

`TakeOver`	#40788h
Indicate a custom key definition	
→	

Expanding the last example on the previous page, a hardkey handler with custom code for two unshifted arrow keys and two right-shifted arrow keys looks like this:

```
::
  kpNoShift #=casedrop
    ::
      DUP#<7 casedrpfls                          Enables primary softkeys
      kcRightShift #=casedrpfls                  Enables ↱
      kcNextRow #=casedrpfls                     Enables [NXT]
      kcLeftArrow ?CaseKeyDef
        :: TakeOver do left key ;                Process ◄
      kcRightArrow ?CaseKeyDef
        :: TakeOver do right key ;               Process ►
      issue error beep for remaining invalid keys
    ;
  kpRightShift #=casedrop
    ::
      kcRightShift #=casedrpfls                  Enables ↱
      kcOn #=casedrpfls                          Enables ↱ [OFF]
      kcLeftArrow ?CaseKeyDef
        :: TakeOver do left key ;                Process ↱ ◄
      kcRightArrow ?CaseKeyDef
        :: TakeOver do right key ;               Process ↱ ►
      issue error beep for remaining invalid keys
    ;
  2DROP 'DoBadKeyT                               Reject all other planes
;
```

Now all that remains is to generate an invalid key beep for the remaining keys.

Signaling Invalid Keys. If your application does not define the key, you may wish to prevent the standard definition from being executed and generate an invalid key beep. To do this, you actually define the key to generate an invalid key beep. The object `DoBadKey` is suited for this purpose:

`DoBadKey`	#3FDD1h
Generate a bad key beep and execute `SetDAsNoCh`	
→	

As you build your key handlers, the following objects become useful:

`'DoBadKey`	#3FDFEh
Places a pointer to `DoBadKey` on the stack	
→ `DoBadKey`	
`'DoBadKeyT`	#3FE12h
Places a pointer to `DoBadKey` and TRUE on the stack	
→ `DoBadKey TRUE`	

A Complete Hardkey Handler. Expanding the previous example, a complete hardkey handler with custom code for two unshifted arrow keys, two left-shifted arrow keys, and two right-shifted arrow keys, a multi-row softkey menu, and [OFF] looks like this:

```
::
  kpNoShift #=casedrop
    ::
      DUP#<7 casedrpfls              Enables primary softkeys
      kcRightShift #=casedrpfls      Enables [↱]
      kcLeftShift #=casedrpfls       Enables [↰]
      kcNextRow #=casedrpfls         Enables [NXT]
      kcLeftArrow ?CaseKeyDef
        :: TakeOver do left key ;    Process [◄]
      kcRightArrow ?CaseKeyDef
        :: TakeOver do right key ;   Process [►]
      DROP 'DoBadKeyT                Issue invalid key beep
    ;
  kpRightShift #=casedrop
    ::
      kcRightShift #=casedrpfls      Enables [↱]
      kcLeftShift #=casedrpfls       Enables [↰]
      kcNextRow #=casedrpfls         Enables [↱] [NXT]
      kcLeftArrow ?CaseKeyDef
        :: TakeOver do left key ;    Process [↱] [◄]
      kcRightArrow ?CaseKeyDef
        :: TakeOver do right key ;   Process [↱] [►]
      kcOn #=casedrpfls              Enables [↱] [OFF]
      DROP 'DoBadKeyT                Issue invalid key beep
    ;
  kpLeftShift #=casedrop
    ::
      kcRightShift #=casedrpfls      Enables [↱]
      kcLeftShift #=casedrpfls       Enables [↰]
      kcNextRow #=casedrpfls         Enables [↰] [PREV]
      kcLeftArrow ?CaseKeyDef
        :: TakeOver do left key ;    Process [↰] [◄]
      kcRightArrow ?CaseKeyDef
        :: TakeOver do right key ;   Process [↰] [►]
      DROP 'DoBadKeyT                Issue invalid key beep
    ;
  2DROP 'DoBadKeyT                   Reject all other planes
;
```

8.7 Softkey Definitions

A softkey definition can be as simple (an object that is echoed into the command line) or complex (a dynamic label with different actions for different shift planes) as you like. The menu keys for the solver, multiple equation solver, and modes are illustrations of complex menu definitions in the HP 48.

The basic structure of a softkey definition consists of a list where the first object defines the label and the second object defines the actions taken when the key is pressed:

```
{ label_object action_object }
```

The softkey definition in the example POL1 in previous pages is structured just this way:

```
{
  "QUIT"                                 Label text
  :: TakeOver TRUE ' LAM Running STO ;   Key action
}
```

In the following sections we'll describe how the label object and the action object can be structured.

8.7.1 Null Menu Keys

Some menus have blank keys that generate an error beep as their defined action. These keys are used to help distribute labels within the menu row. The object `NullMenuKey` defines a blank key, and can be used in your menu definition as shown in the example POL1 at the beginning of this chapter.

`NullMenuKey`	#3EC71h
Defines a blank menu key	
→	{ *menu definition* }

8.7.2 Softkey Label Objects

A softkey label object may consist of any of the following:

String Any string object may be used as a label. Remember that the small font used for labels is not a fixed-width font, so some words will fit in a label and others won't. In the HP 48G/GX, the left parenthesis character "(" was used for the letter "C" in the input form and choose box "CANCL" menu labels.

8x21 Grob A grob that is 8 rows high and 21 characters wide may be used for the label. Grobs that are not this size will be decompiled into a string and that string will be used for the label.

Secondary A secondary that begins with `TakeOver` is expected to return either of the above — a string or a grob. Utilities first introduced in *Menu Grob Utilities* on page 128 are useful for returning menu label grobs, and will be illustrated below. These are sometimes called *takeover secondaries*.

Anything Else Any other object is decompiled to string form and that string is used for the label.

Dynamic Labels. The third case mentioned above — a secondary beginning with `TakeOver` — provides the most flexibility for the label portion of a softkey definition. The secondary can do anything it likes as long as it follows two basic rules:

- The stack *must* remain as it was found. If your secondary needs to know which position in the menu is being displayed, the object `INDEX@` may be used to return a bint index from 1 to 6.
- The secondary must return a string or a 8x21 grob.

The example program POL2 provides a concise demonstration of a dynamic label. When this program is running, the first softkey enables a toggle of user flag 1. The object `?DispStatus` is used to show the system status, illustrating the action of the softkey.

This example has a short menu definition — just one key. The [ON] key terminates the program (instead of the `|QUIT|` softkey in POL1).

```
POL2 218.5 Bytes Checksum #7D32h
( → )
```

```
DEFINE kpNoShift ONE
DEFINE kcOn FORTYFIVE
::
  0LASTOWDOB!                                Clear saved command name
  CK0NOLASTWD                                No arguments
  RECLAIMDISP                                Clear display
  FALSE                                      Exit flag
  ' LAM Running
  ONE DOBIND                                 Create temporary envitonment
  ' ::                                       Display action
    DA3OK? ?SKIP :: DispMenu.1 SetDA3Valid ;   Display menu if not done already
    ?DispStatus                              Display the status area
  ;
  ' ::                                       Hardkey handler:
      kpNoShift #=casedrop                   Process primary key plane:
        ::
          DUP#<7 casedrpfls                  Enable softkeys
          kcOn ?CaseKeyDef                   Process [ON] key
            :: TakeOver TRUE LAM Running STO ;
          DROP 'DoBadKeyT                    Reject all other keys
        ;
      2DROP 'DoBadKeyT                       Reject all other planes
    ;
  TRUE                                       Enable softkeys
  TRUE                                       Reject user key definitions
  {                                          Softkey menu:
    {
      ::                                     Label secondary
        TakeOver
        "1" ONE TestUserFlag                 Test user flag 1
        Box/StdLabel                         Use test result to create label
      ;
      ::                                     Key action:
        TakeOver
        ONEONE TestUserFlag                  Test user flag
        ITE ClrUserFlag SetUserFlag          Toggle flag state
        SetDA1Bad SetDA3Bad                  Signal to redraw status and menu
      ;
    }
  }
  ONEFALSE                                   Display 1st menu row, no suspend
  ' LAM Running                              Exit object
  ' ERRJMP                                   Error handler
  ParOuterLoop                               Run the POL
  ABND                                       Discard temporary environment
  ClrDAsOK                                   Signal to redraw the display
;
```

8.7.3 Softkey Action Object

The action object may define actions for the primary, left-shift, and right-shift planes. Action objects consist of a takeover secondary, or a list containing two or three takeover secondaries, as follows:

```
:: TakeOver ... ;       Action object for the primary plane

{
  :: TakeOver ... ;     Action object for the primary plane
  :: TakeOver ... ;     Action object for the left-shift plane
}

{
  :: TakeOver ... ;     Action object for the primary plane
  :: TakeOver ... ;     Action object for the left-shift plane
  :: TakeOver ... ;     Action object for the right-shift plane
}
```

Remember: The hardkey handler *must* enable the shift planes for the shift-action objects to work.

The example POL3 below defines a one-key menu. The key definition consists of a string for the label object and an action object list defining primary, left-, and right-shift actions. Notice that each action begins with the object `TakeOver`.

```
POL3 343.5 Bytes Checksum #355h
( → )
DEFINE kpNoShift     ONE
DEFINE kpLeftShift   TWO
DEFINE kpRightShift  THREE
DEFINE kcLeftShift   THIRTYFIVE
DEFINE kcRightShift  FORTY
DEFINE kcOn          FORTYFIVE
::
  0LASTOWDOB! CK0NOLASTWD                              Clear protection word, no
                                                       arguments
  RECLAIMDISP ClrDA1IsStat                             Clear display, suspend clock
  FALSE ' LAM Running ONE DOBIND                       Exit flag
  ' :: DA3OK? ?SEMI DispMenu.1 SetDA3Valid ;           Display action
  ' ::                                                 Hardkey handler:
    kpNoShift #=casedrop                               Primary plane
      ::
        DUP#<7 casedrpfls
        kcLeftShift #=casedrpfls
        kcRightShift #=casedrpfls
        kcOn ?CaseKeyDef
          :: TakeOver TRUE ' LAM Running STO ;
        DROP 'DoBadKeyT
      ;
    kpLeftShift #=casedrop                             Left-shift plane
      ::
        DUP#<7 casedrpfls
        kcLeftShift #=casedrpfls
        kcRightShift #=casedrpfls
        DROP 'DoBadKeyT
      ;
    kpRightShift #=casedrop                            Right-shift plane
      ::
        DUP#<7 casedrpfls
        kcLeftShift #=casedrpfls
        kcRightShift #=casedrpfls
        kcOn #=casedrpfls
        DROP 'DoBadKeyT
      ;
    2DROP 'DoBadKeyT
```

```
  ;
  TRUE TRUE                                         Key flags
  {                                                 Softkey menu
    {
      "KEY"
      {
        :: TakeOver "Primary" DISPROW3 VERYSLOW DOCLLCD ;
        :: TakeOver "Left-Shift" DISPROW4 VERYSLOW DOCLLCD ;
        :: TakeOver "Right-Shift" DISPROW5 VERYSLOW DOCLLCD ;
      }
    }
  }
  ONEFALSE
  ' LAM Running
  ' ERRJMP
  ParOuterLoop
  ABND
  ClrDAsOK
;
```

8.8 The POL Error Trap Object

In the previous POL examples we have specified a standard error trap by leaving a pointer to ERRJMP on the stack. Here we illustrate an error trap designed to detect and handle a specific class of errors that occur while a key definition is being executed and pass remaining errors off to the system outer loop.

Note that this error trap does *not* handle errors generated during the execution of the display object.

The example POL4 below displays a value and its inverse. The key [+] is defined to increment the value and [-] is defined to decrement the value. When the value is zero, the operation 1/*value* generates an error, which is handled by the error object. The softkey |→ERR| generates an error that the error object does not recognize and passes on. The program ends when ON is pressed.

The error handler illustrated in POL4 takes advantage of the numbering of the error messages in the HP 48. Any error that is floating-point related is in the #300h range (see the appendix *Messages* on page 243). The error handler divides the error number by #100h and discards the remainder, so the result will be 3 if a floating point error has occurred. If the error is not a floating point error, the error is passed to the system outer loop with ERRJMP, otherwise the error handler displays the appropriate text.

This technique is similar to the scheme used by the HP 48 DRAW command, which is the core of the plotting system. Notice that when you plot a function like SIN(1/X) no error is generated when X=0.

POL4 555 Bytes Checksum #A4C4h
(→)

```
DEFINE kpNoShift  ONE
DEFINE kcOn       FORTYFIVE
DEFINE kcMinus    FORTYFOUR
DEFINE kcPlus     FORTYNINE
::
  OLASTOWDOB! CK0NOLASTWD                      Clear protection word, no
                                               arguments
  RECLAIMDISP ClrDA1IsStat                     Clear display, suspend clock
  ' ::                                         Display object for key handlers
    "Value: " LAM Value EDITDECOMP$ &$ DISPROW3
    "Result: " LAM Result EDITDECOMP$ &$ DISPROW4
  ;
  %1 %1                                        Initial result and initial value
  FALSE                                        Exit flag
```

```
  ' LAM DoDisplay
  ' LAM Result
  ' LAM Value
  ' LAM Running
  FOUR DOBIND                                         Create temporary environment
  LAM DoDisplay EVAL                                  Initial display of value and result
  ' :: DA3OK? ?SEMI DispMenu.1 SetDA3Valid ;          Display handler
  ' ::                                                Hardkey handler:
      kpNoShift #=casedrop
        ::
          DUP#<7 casedrpfls                           Enable softkeys
          kcMinus ?CaseKeyDef                         [-]
            :: TakeOver
               LAM Value %1- DUP ' LAM Value STO %1/
               ' LAM Result STO LAM DoDisplay EVAL
            ;
          kcPlus ?CaseKeyDef                          [+]
            :: TakeOver
               LAM Value %1+ DUP ' LAM Value STO %1/
               ' LAM Result STO LAM DoDisplay EVAL
            ;
          kcOn ?CaseKeyDef                            [ON]
            :: TakeOver
               TRUE ' LAM Running STO
            ;
          DROP 'DoBadKeyT                             Reject other keys
        ;
      2DROP 'DoBadKeyT                                Reject other planes
  ;
  TRUE TRUE                                           Key control flags
  {                                                   Softkey menu
    { "\8DERR" :: TakeOver "Unhandled Error" DO$EXIT ; }
  }
  ONEFALSE                                            Display 1st menu row, no suspend
  ' LAM Running                                       Exit object
  ' ::                                                Error handler:
    ERROR@                                              Recall the error number
    # 100 #/ SWAPDROP THREE #<> case ERRJMP             ERRJMP if not floating-point
    ERRORCLR                                            Clear the error number
    "Value: " LAM Value EDITDECOMP$ &$ DISPROW3         Display the value
    "Result: Undefined" DISPROW4                        Display "Undefined" for result
  ;
  ParOuterLoop                                        Run the POL
  ABND                                                Discard temporary environment
  ClrDAsOK                                            Signal to redraw the display
;
```

8.9 POL Utilities

There are times when using constituent components of the object ParOuterLoop is either appropriate or required. ParOuterLoop is written as follows:

```
::
  POLSaveUI              Save the current user interface
  ERRSET                 Increment the protection word
    ::
      POLSetUI           Set the application user interface
      POLKeyUI           Process keys
    ;
  ERRTRAP POLResUI&Err   If an error occurs, restore the old user interface and ERRJMP
  POLRestoreUI           Restore the user interface
;
```

There are two basic reasons for using these utilities individually:

- An application can use null-named temporary variables, saving memory and execution time.
- An application that uses or interchanges between several POLs can save the execution overhead associated with saving and restoring the original user interface.

`POLSaveUI`	#389BCh
Save the current user interface →	
`POLSetUI`	#38A64h
Establish the parameters for the POL *Parameters for* `ParOuterLoop` →	
`POLKeyUI`	#38AEBh
Run the POL →	
`POLResUI&Err`	#38B77h
Standard POL error handler →	
`POLRestoreUI`	#38B90h
Restore the user interface saved by `POLSaveUI` →	

There are many possible ways to use these utilities. The browser engine from the equation library (described in *Graphic User Interfaces* on page 169) presumes that the calling application has saved the user interface and only calls `POLSetUI` and `POLKeyUI`.

One possible structure for an application using these utilities looks like this:

```
::
  OLASTOWDOB! CKONOLASTWD          Clear protection word, no arguments
  RECLAIMDISP ClrDA1IsStat         Claim the display
  POLSaveUI                        Save the user interface
  ERRSET                           Increment the protection word
    ::                             Process keys
      ONE
      TRUE
      ' LAM InterfaceIndex         Variable to store the interface index
      ' LAM AppRunning             Master "running" variable
      TWO DOBIND
      BEGIN
        LAM AppRunning
      WHILE
        {                          List of interface parameters
          { POL parameters for interface 1 }
          { POL parameters for interface 2 }
          { POL parameters for interface 3 }
        }
        LAM InterfaceIndex         Recall index
        NTHCOMPDROP                Extract interface
        INCOMPDROP                 Put parameters on the stack
        POLSetUI                   Set the user interface
        POLKeyUI                   Run the user interface
      REPEAT
    ;
  ERRTRAP POLResUI&Err             Master error trap
  POLRestoreUI                     Restore the user interface
;
```

This application uses an index stored in the local variable `InterfaceIndex` to decide which interface to run as long as the flag stored in `AppRunning` is TRUE. In the structure, the key handlers are responsible for storing a new index value into `InterfaceIndex` when signaling a switch to another interface, and storing FALSE into `AppRunning` when the entire application should terminate.

8.10 Menu Utilities

The utilities `InitMenu` and `InitMenu%` are useful for applying a new menu definition to the current environment. In combination with objects like `DispMenu` and `DispMenu.1`, you can initialize and display a menu. See also *Menu Area Control* on page 114.

`DispMenu`	#3A1E8h
Displays the current menu and freezes the menu display line	
→	
`DispMenu.1`	#3A1FCh
Displays the current menu	
→	
`InitMenu`	#40F86h
Establishes a menu	
{ *menu definition* } →	
`InitMenu%`	#41679h
Displays a built-in menu based on a menu number	
% →	

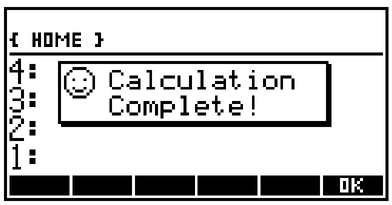
HP 48G/GX Menu Numbers					
#	**Menu Name**	**#**	**Menu Name**	**#**	**Menu Name**
0	LAST Menu	40	PRG OUT	80	↰ SOLVE TVM SOLVR
1	CST	41	PRG RUN	81	↰ PLOT
2	VAR	42	↱ UNITS	82	↰ PLOT PTYPE
3	MTH	43	↱ UNITS LENG	83	↰ PLOT PPAR
4	MTH VECTR	44	↱ UNITS AREA	84	↰ PLOT 3D
5	MTH MATR	45	↱ UNITS VOL	85	↰ PLOT 3D PTYPE
6	MTH MATR MAKE	46	↱ UNITS TIME	86	↰ PLOT 3D VPAR
7	MTH MATR NORM	47	↱ UNITS SPEED	87	↰ PLOT STAT
8	MTH MATR FACTR	48	↱ UNITS MASS	88	↰ PLOT STAT PTYPE
9	MTH MATR COL	49	↱ UNITS FORCE	89	↰ PLOT STAT $\sum$PAR
10	MTH MATR ROW	50	↱ UNITS ENRG	90	↰ PLOT STAT $\sum$PAR MODL
11	MTH LIST	51	↱ UNITS POWR	91	↰ PLOT STAT DATA
12	MTH HYP	52	↱ UNITS PRESS	92	↰ PLOT FLAG
13	MTH PROB	53	↱ UNITS TEMP	93	↰ SYMBOLIC
14	MTH REAL	54	↱ UNITS ELEC	94	↰ TIME
15	MTH BASE	55	↱ UNITS ANGL	95	↰ TIME ALARM
16	MTH BASE LOGIC	56	↱ UNITS LIGHT	96	↰ STAT
17	MTH BASE BIT	57	↱ UNITS RAD	97	↰ STAT DATA
18	MTH BASE BYTE	58	↱ UNITS VISC	98	↰ STAT $\sum$PAR
19	MTH FFT	59	↰ UNITS	99	↰ STAT $\sum$PAR MODL
20	MTH CMPL	60	PRG ERROR IFERR	100	↰ STAT 1VAR
21	MTH CONS	61	PRG ERROR	101	↰ STAT PLOT
22	PRG	62	↰ CHARS	102	↰ STAT FIT
23	PRG BRCH	63	↰ MODES	103	↰ STAT SUMS
24	PRG IF	64	↰ MODES FMT	104	↰ IO
25	PRG CASE	65	↰ MODES ANGL	105	↰ IO SRVR
26	PRG START	66	↰ MODES FLAG	106	↰ IO IOPAR
27	PRG FOR	67	↰ MODES KEYS	107	↰ IO PRINT
28	↰ EDIT	68	↰ MODES MENU	108	↰ IO PRINT PRTPA
29	PRG DO	69	↰ MODES MISC	109	↰ IO SERIA
30	↰ SOLVE ROOT SOLVR	70	↰ MEMORY	110	↰ LIBRARY
31	PRG WHILE	71	↰ MEMORY DIR	111	↰ LIBRARY PORTS
32	PRG TEST	72	↰ MEMORY ARITH	112	↰ LIBRARY
33	PRG TYPE	73	↰ STACK	113	↰ EQ LIB
34	PRG LIST	74	↰ SOLVE	114	↰ EQ LIB EQLIB
35	PRG LIST ELEM	75	↰ SOLVE ROOT	115	↰ EQ LIB COLIB
36	PRG LIST PROC	76	↰ SOLVE DIFFEQ	116	↰ EQ LIB MES
37	PRG GROB	77	↰ SOLVE POLY	117	↰ EQ LIB UTILS
38	PRG PICT	78	↰ SOLVE SYS		
39	PRG IN	79	↰ SOLVE TVM		

HP 48S/SX Menu Numbers			
#	**Menu Name**	**#**	**Menu Name**
0	LAST Menu	30	SOLVE SOLVR
1	CST	31	↰ PLOT
2	VAR	32	PLOT PTYPE
3	MTH	33	PLOT PLOTR
4	MTH PARTS	34	↰ ALGEBRA
5	MTH PROB	35	↰ TIME
6	MTH HYP	36	TIME ADJST
7	MTH MATRX	37	TIME ALRM
8	MTH VECTR	38	TIME ALRM RPT
9	MTH BASE	39	TIME SET
10	PRG	40	↰ STAT
11	PRG STK	41	STAT MODL
12	PRG OBJ	42	↰ UNITS
13	PRG DISP	43	UNITS LENG
14	PRG CTRL	44	UNITS AREA
15	PRG BRCH	45	UNITS VOL
16	PRG TEST	46	UNITS TIME
17	PRINT	47	UNITS SPEED
18	I/O	48	UNITS MASS
19	I/O SETUP	49	UNITS FORCE
20	↰ MODES	50	UNITS ENRG
21	↱ MODES	51	UNITS POWR
22	↰ MEMORY	52	UNITS PRESS
23	↱ MEMORY	53	UNITS TEMP
24	↰ LIBRARY	54	UNITS ELEC
25	LIBRARY PORT 0	55	UNITS ANGL
26	LIBRARY PORT 1	56	UNITS LIGHT
27	LIBRARY PORT 2	57	UNITS RAD
28	↰ EDIT	58	UNITS VISC
29	↰ SOLVE	59	↱ UNITS

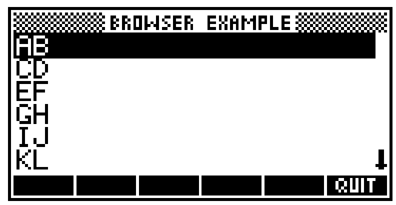
Chapter 9

Graphical User Interfaces

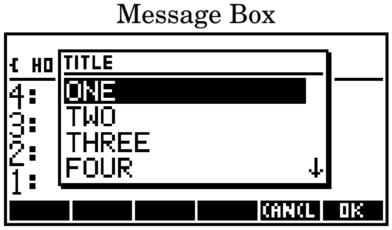
The HP 48G/GX calculators are characterized in part by the introduction of three new basic user interface tools — message boxes, choose boxes, and input forms. The Equation Library, originally distributed on a plug-in card for the HP 48S/SX, is now built into the HP 48G series and has its own browser.



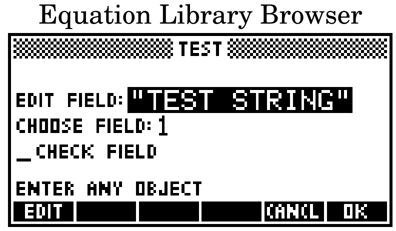
Message Box



Equation Library Browser



Choose Box



Input Form

In this chapter we introduce the basic interface to each of these components. Going beyond the Parameterized Outer Loop, the choose boxes and input forms require a blizzard of stack arguments. We suggest you read this chapter in chronological order, since each part builds upon the previous part. Also, you might want to back up your HP 48 memory prior to starting your explorations.

Note: The objects described in this chapter are only available in the HP 48G/GX.

EXTERNAL Declarations in Examples. Some examples have EXTERNAL declarations at the beginning for each object that is referenced by a rompointer (XLIB name) instead of a hard address. This EXTERNAL declaration is used by the HP RPLCOMP.EXE compiler. Other tools may have different methods of indicating a rompointer.

Objects Used in Examples. In this chapter we presume you've read and understood the previous chapters fairly well. We'll be using objects and techniques described earlier, and the comments in the examples will pertain more to the technique being described and less to the actions of individual objects. You may wish to refer to previous descriptions of some of the objects used to fully understand the details of some of the examples.

9.1 Message Boxes

A message box is useful for presenting a message, waiting for the user to read it, and moving on. This object, called `DoMsgBox`, is the HP 48G/GX's tool for providing the dreaded "Press Any Key To Continue" style prompt that computers are famous for. In this case, the message box engine is terminated by pressing [OK], [ENTER], or [ON]. `DoMsgBox` will save and restore the display, so the calling application need not worry about the display.

The message box engine attempts to provide some basic text formatting within the box, so you don't have to worry about where word breaks will occur. Two bints specify the minimum and maximum character widths of the box, and adjusting these gives you a little more control over the appearance of the message box.

9.1.1 Message Box Parameters

The parameters for `DoMsgBox` are defined as follows:

`DoMsgBox`	#000B1h	G/GX XLIB 177 0
Displays a message box with a graphics object		
message #maxwidth #minwidth grob menuobject → TRUE		

"message" A string containing the message you wish to display. Carriage-returns may be embedded to force line breaks.

#maxwidth A bint specifying the maximum character width of each text line in the message box. Message boxes use only the medium (5x7) font.

#minwidth A bint specifying the minimum number of characters to be displayed before an automatic word break is allowed.

grob A graphics object to be displayed in the upper-left corner of the message box. If you don't want to include a grob, specify the bint MINUSONE as the grob. The grob `grobAlertIcon` is handy for use in message boxes:

`grobAlertIcon`	#850B0h	G/GX XLIB 176 133
The message box alert icon		
	→ grob	

menuobject An object which, when evaluated, produces a message box menu. This is usually specified as `MsgBoxMenu`, which is function 2 in library 177:

`MsgBoxMenu`	#020B1h	G/GX XLIB 177 2
The message box menu		
	→ {menu}	

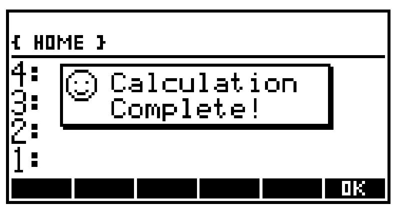
`DoMsgBox` returns the flag TRUE. You may wish to try different values for the character widths to adjust where automatic word breaks occur. Neither value should exceed 15. Remember to leave room for the grob.

9.1.2 Message Box Example

The following example uses an 11x11 grob for an icon in a message box.

MBOX 100 Bytes Checksum #D7D8h
(→)

```
EXTERNAL DoMsgBox               Declares DoMsgBox is referenced by a rompointer
EXTERNAL MsgBoxMenu             Declares MsgBoxMenu is referenced by a rompointer
::
  0LASTOWDOB! CK0NOLASTWD       Clear the protection word, no arguments
  "Calculation Complete!"       Message text
  TWELVE                        Maximum character width
  TEN                           Minimum character width
ASSEMBLE                        Grob
  CON(5)  =DOGROB
  REL (5) end
  CON(5)  11
  CON(5)  11
  NIBHEX  8F00401020201040
  NIBHEX  9840104010409840
  NIBHEX  272040108F00
end
RPL
  ' MsgBoxMenu                  Message box menu
  DoMsgBox                      Execute the message box
  DROP                          Drop the returned flag
  ClrDAsOK                      Signal to redraw the display
;
```



9.2 Equation Library Browser

The browser used by the equation library dates back to the HP Solve Equation Library card originally sold for the HP 48SX. When the Equation Library was built into the HP 48G/GX, the browser was not replaced by the new choose box engine (described later in this chapter).

To use the browser, create a shell using Parameterized Outer Loop utilities that has the following structure:

```
::
  ...
  POLSaveUI              Save the user interface
  ERRSET                 Increment the protection word
    ::
      ...
      BRbrowse           Call the browser
      ...
    ;
  ERRTRAP POLResUI&Err   If an error occurs, restore the old user interface and ERRJMP
  POLRestoreUI           Restore the user interface
  ...
;
```

9.2.1 Browser Parameters

The browser requires eight parameters and returns nothing to the stack. The browser can only be terminated by executing the object `BRdone`.

`BRbrowse`	#100E0h	G/GX XLIB 224 16
Browse a list		
{menu} $title {key defs} #first_row #focus_pos {data} :: data secondary ; {speed} →		
`BRdone`	#130E0h	G/GX XLIB 224 19
Terminate the browser		
→		

The parameters for `BRbrowse` are specified as follows:

{menu} A softkey menu, specified the same way as a menu for any Parameterized Outer Loop.

$title A string for the title bar. If this string is null, seven rows of data will be displayed, otherwise the title bar will be displayed with six rows of data.

{ [ENTER] [ON] } A list containing a procedure to execute when [ENTER] is pressed and a procedure to execute when [ON] is pressed. These procedures take no input parameters and may return anything.

#first_row A bint specifying the index of the first data item to be displayed.

#focus_pos A bint specifying which data item is highlighted first.

{data} A list containing the items to display. If the data secondary is going to return the data from another location, this list may be empty.

:: data_secondary ; A secondary that accepts the data list and a bint and returns either the number of data items (if the bint is zero) or a string (if the bint is non-zero):

{data} ZERO	→	#number_of data_items
{data} #index	→	$item

{speed} A speed table for alpha searches. The table consists of a list of 26 index bints corresponding to the letters A – Z. If the user presses [α] [D], the fourth bint is tested. If non-zero, this bint is assumed to be the index of the first item in the data list that starts with 'D'. If the speed table is an empty list, it is not used.

9.2.2 Active Browser Keys

While the browser is active, the following keys are active:

[▲] [▼] The arrow keys move the highlight up or down one row.

[↰] [▲] or

[↰] [▼] Pressing [↰] and an arrow key moves the highlight to the bottom of the screen or to the bottom of the next screen if the highlight is already at the bottom of the screen.

[↱] [▲] or

[↱] [▼] Pressing [↱] and an arrow key moves the highlight to the beginning or end of the data list.

[α] Press [α] and a letter to move to the next item starting with that letter.

[ENTER] Executes the supplied [ENTER] procedure.

[ON] Executes the supplied [ON] procedure.

|MENU| Executes a softkey definition.

9.2.3 Browser Support Objects

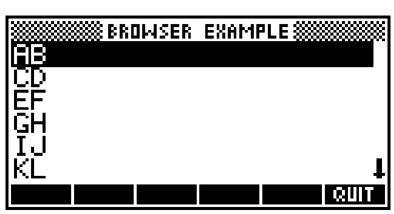
While the browser is active, the following objects are available for use by key definitions:

`BRDispItems`	#450E0h	G/GX XLIB 224 69	Displays the items for each row and the more-data arrows		→	
`BRGetItem`	#530E0h	G/GX XLIB 224 83	Gets the item for the specified index	#index	→	$
`BRinverse`	#490E0h	G/GX XLIB 224 73	Inverts the highlight		→	
`BRoutput`	#120E0h	G/GX XLIB 224 18	Recall the index of the highlighted data item and the index of the first row		→	#first_row #focus_pos
`BRRclC1`	#180E0h	G/GX XLIB 224 24	Recall the data list		→	{ data }
`BRRclCurRow`	#170E0h	G/GX XLIB 224 23	Recall the index of the highlighted data item		→	#focus_pos
`BRStoC1`	#030E0h	G/GX XLIB 224 24	Store the data list (must be the same length as previous list)	{ data }	→	
`BRViewItem`	#520E0h	G/GX XLIB 224 82	Display the highlighted item using the full display, wait for a keystroke. Respects linefeed breaks if present. Redraws browser display after keystroke.		→	

9.2.4 Browser Example

The program BRW1 displays a short list using the browser and returns a string indicating which key terminated the browser. If the browser was terminated by pressing [ENTER] the highlighted data item is returned.

```
BRW1 265 Bytes Checksum #69DFh
( → "ON")                                Terminated by pressing [ON]
( → "QUIT")                              Terminated by pressing |QUIT|
( → $item "ENTER")                       Terminated by pressing [ENTER]
EXTERNAL BRbrowse
EXTERNAL BRdone
EXTERNAL BRRclC1
EXTERNAL BRRclCurRow
::
  0LASTOWDOB! CK0NOLASTWD"               Clear saved command name, no arguments
  ClrDA1IsStat RECLAIMDISP               Claim the display
  POLSaveUI                              Save the current user interface
  ERRSET                                 Increment the protection word
  ::
    {                                    Menu for the browser
      NullMenuKey
      NullMenuKey
      NullMenuKey
      NullMenuKey
      NullMenuKey
      {
        "QUIT"                           Softkey label
        :: TakeOver "QUIT" BRdone ;      Return "QUIT", signal to terminate the browser
      }
    }
    "BROWSER EXAMPLE"                    Browser title
    {                                    Hardkey list:
      ::                                 [ENTER]
        BRRclC1 BRRclCurRow NTHCOMPDROP  Returns the highlighted data item
        "ENTER"                          Returns the string "ENTER"
        BRdone                           Signal to terminate the browser
      ;
      ::                                 [ON]
        "ON"                             Return the string "ON"
        BRdone                           Signal to terminate the browser
      ;
    }
    ONE ONE                              First displayed row and highlighted row
    { "AB" "CD" "EF" "GH" "IJ" "KL" "MN" "OP" }  Data list
    ' ::                                 Data secondary
        ZERO #=casedrop LENCOMP          Return length of data list if index is 0
        NTHCOMPDROP                      Otherwise return the item
    ;
    NULL{}                               No speed list
    BRbrowse                             Display the browser
  ;
  ERRTRAP POLResUI&Err                   If error occurs, restore old interface and error
  POLRestoreUI                           Restore the old interface
  ClrDAsOK                               Signal to redraw the display
;
```

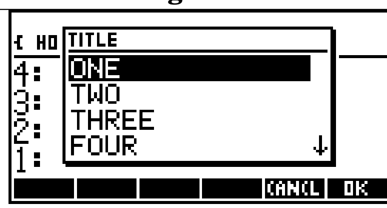
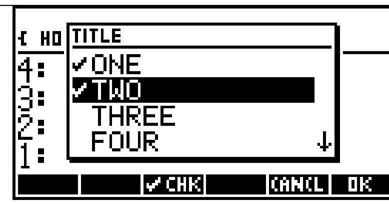
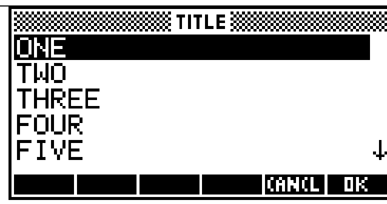
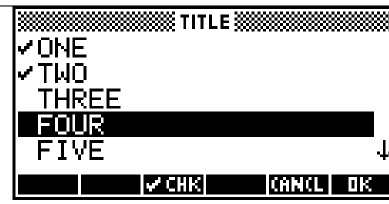


9.3 Choose Boxes

A choose box lets the user select one or more items from a series of choices or view a series of choices. This section describes the basic types of choose boxes and how to customize them.

9.3.1 Choose Box Styles

There are three basic types of choose boxes — *single-pick*, *multi-pick*, and *view-only*. A single-pick choose box lets the user choose a single item from a list of choices. The multi-pick choose box lets the user specify one or more choices with check marks. A choose box can occupy either a shadow-box within the display or the whole display:

Choose Box Style Options		
	Single-Pick	**Multi-Pick**
Partial Screen		
Full Screen		

When a choose box is active, the following keys are defined:

[▲] Moves the highlight up one row.

[▼] Moves the highlight down one row.

[α] *letter* Moves the highlight to the next row beginning with *letter*.

[↱] [▲] Jumps the highlight up to the first choice.

[↰] [▲] Displays the previous page of choices.

[↰] [▼] Displays the next page of choices.

[↱] [▼] Jumps the highlight down to the last choice.

[↱] [OFF] Turns off the HP 48.

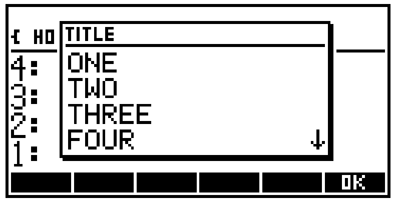
[+/-] Shortcut key for checking an item.

| CHK | Checks the highlighted item in a multi-pick choose box.

|CANCL| or [ON] Cancels the choose box.

| OK | or [ENTER] Terminates the choose box, selecting the highlighted or checked item(s). In a multi-pick choose box, selects the highlighted item if no items are checked.

Any of the above choose box styles may also be used as a display-only viewing device, where no highlight bar is shown:



When a view-only choose box is active, the arrow keys scroll the list, ⇨ [OFF] turns the HP 48 off, and [ON], [ENTER], and OK terminate the choose box.

9.3.2 Choose Box Parameters

Choose boxes are specified both by stack arguments supplied to the object `Choose` and by responses to various messages generated by the choose box engine. The object `Choose` produces the choose box, using five stack arguments as input:

`Choose`	#000B3h	G/GX XLIB 179 00
Display a choose box		
Msg-handler TitleOb DecompOb { *choices* } #FocusPos → ob TRUE		*Single-pick input accepted*
Msg-handler TitleOb DecompOb { *choices* } #FocusPos → { ob_1 ... ob_n } TRUE		*Multi-pick input accepted*
Msg-handler TitleOb DecompOb { *choices* } #FocusPos → FALSE		*Cancelled or view-only*

Message Handler The message handler provides opportunities to customize the choose box and react to specific events by responding to messages.

Title Object An object which, when evaluated, produces a string for the choose box title. If a null-length string is provided, no title will be displayed, title related messages will not be generated, and an extra row will be available for displaying choices.

Decompile Object Specifies the manner in which each choice will be displayed.

{ *choices* } A list of the choices. The choices must all have the same structure. Typical examples include:

- A bint specifying a built-in message number
- An object
- A list containing two objects, one of which will be used to display the choice, the other of which is associated with the first for post-choosebox evaluation

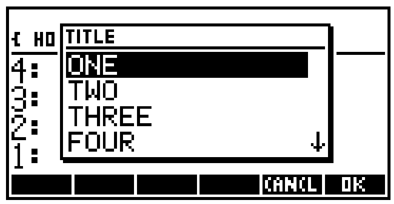
#FocusPos The focus position is the position of the highlight within the data list. A bint specifies the initial focus position. If the bint is zero, the choose box displays a view-only list.

The message handler, decompile object, and data list will be described further below.

Example: We begin by looking at a simple choose box. CHS1 displays a default choose box showing a list of six string objects:

```
CHS1 101 Bytes Checksum #B027h
( → )
```

```
EXTERNAL Choose                  Declare Choose a rompointer
::
  AtUserStack                    Clear saved command name, no arguments
  ' DROPFALSE                    Message handler
  "Title"                        Choose box title string
  ONE                            Decompile format
  {                              List of choices
    "ONE" "TWO" "THREE"
    "FOUR" "FIVE" "SIX"
  }
  ONE                            Initial focus position
  Choose                         Display the choose box
  COERCEFLAG                     Exit, converting the result flag to %1 or %0
;
```



9.3.3 Choose Box Message Handler

At various times during the execution of the choose box, the choose box engine sends a message to the message handler. If the message handler chooses not to handle the message, the default behavior related to that message will occur. If the message handler does handle the message, the default behavior does not happen. If you don't plan to handle any messages, then the object `DROPFALSE` is all that's needed, as shown above.

A message arrives at the message handler in the form of a binary integer indicating the message type with optional stack parameters. The message handler is expected to return TRUE if the message was handled, along with any required results on the stack, or FALSE if the message was not handled.

A message handler has the following stack diagram:

<passed objects> #message	→	<*returned objects*> TRUE
<passed objects> #message	→	<passed objects> FALSE

The following message handler specifies a full-screen multi-pick choose box by handling messages 60 and 61:

```
::
  SIXTY #=casedrop :: TRUE TRUE ;       Handle message 60
  SIXTYONE #=casedrop :: TRUE TRUE ;    Handle message 61
  DROPFALSE                             Ignore other messages
;
```

There are many messages, but the messages most likely to be of interest are listed below:

Message Purpose			Decimal message number
	Input arguments	→	Objects returned by the handler

Choose Box Size			60
		→	TRUE *Full screen choose box*
		→	FALSE *Partial screen choose box*

`Pick Type`			61
	→	TRUE *Multi-pick*	
	→	FALSE *Single-pick*	
`Item Count`			62
	→	#number_of_items_in_list	
`Title Grob`			69
	→	grob	
`Title String`			70
	→	$title	
`Item String`			80
#item_index	→	$item_string	
`Item Grob`			81
#item_index	→	grob	
Note: *Item grob may need to have standard choose item width (91 or 131)*			
`Choose Box Menu`			83
	→	{ menu }	
`Pick Event`			86
	→		
`|CANCL|Key Event`			91
	→	FALSE *Cancel not allowed*	
	→	TRUE *Cancel allowed*	
`| OK |Key Event`			96
	→	FALSE *OK not allowed*	
	→	TRUE *OK allowed*	

Note that you might want to get control when an event happens, but still want the default action to take place. To do this, preserve the passed objects and return FALSE, indicating that you "didn't handle the message".

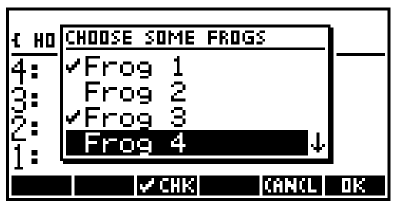
While the choose box is active, null-named temporary variables contain information of interest:

6GETLAM	→	#highlight_row_number
7GETLAM	→	#row_height (pixels)
8GETLAM	→	#row_width (pixels)
12GETLAM	→	#item_count
15GETLAM	→	{ *list of picked indices* }
18GETLAM	→	#index_of highlighted_item
19GETLAM	→	{ *choice_list* }

Example. To introduce some uses of message handling, the message handler in CHS2 specifies the choose box type and choices via the message handler.

```
CHS2 121 Bytes Checksum #28EDh
( → %0 )
( → { choices } %1 )
```

```
EXTERNAL Choose
::
  AtUserStack                        Clear saved command name, no arguments
  ' ::                               Message handler
    SIXTYONE #=casedrop TrueTrue     Specify multi-pick choose box
    SIXTYTWO #=casedrop :: NINE TRUE ;  Specify nine choices
    80 #=casedrop                    Create the string for each choice:
      ::
        UNCOERCE EDITDECOMP$         Convert index bint into real and decompile it
        "Frog " SWAP&$               Prepend frog string
        TRUE                         Signal event handled
      ;
    DROP FALSE                       Do not handle other messages
  ;
  "CHOOSE SOME FROGS"                Title string
  ONE                                Decompile object (not used in this example)
  NULL{}                             Null data list
  ONE                                Initial focus position
  Choose COERCEFLAG                  Run the choose box, then exit, converting flag
;
```



This example will be expanded at the end of this chapter with a customized menu and a dynamic title — see CHS6 on page 182.

9.3.4 Decompile Objects

The decompile object controls the manner in which each item is displayed, has the stack diagram (ob →$), and may be specified three ways:

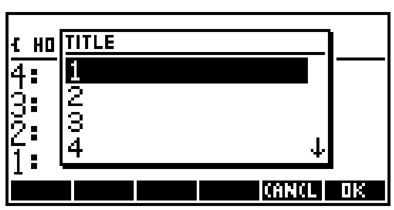
- A pointer to an object that creates a string representation of a choice, like `EDITDECOMP$`
- A secondary that creates a string representation of a choice, like `:: CARCOMP EDITDECOMP$ ;`
- A bint specifying the decompile procedure

The binary integer specification uses specific bits to encode the decompile procedure. These bits control the decompile format, which part of a composite choice to decompile, and whether only the first character should be returned.

Bit	Interpretation
0	No decompilation — expects a string and displays the contents without quote marks
1	Decompile objects as they would appear on the stack (uses the user's numeric display format settings)
2	Decompile objects as they would appear in the editline (uses STD format for numbers)
3	Return only the first character of the string
4	Extract and display the first object of a composite
5	Extract and display the second object of a composite

Example. A bint with the decimal value 36 is supplied as the decompile object for CHS3. Each choice object is actually a list. Bit 2 is set, specifying that objects should be decompiled using STD format. Bit 5 is set, specifying that the second object in the choice list should be decompiled and displayed.

```
CHS3 146 Bytes Checksum #D930h
( → %0 )
( → choice %1 )
EXTERNAL Choose
::
  AtUserStack                Clear saved command name, no arguments
  ' DROPFALSE                Message handler
  "Title"                    Title string
  THIRTYSIX                  Decompile object
  {                          Data list
    { "ONE" %1 }
    { "TWO" %2 }
    { "THREE" %3 }
    { "FOUR" %4 }
    { "FIVE" %5 }
    { "SIX" %6 }
  }
  ONE                        Initial focus position
  Choose                     Run the choose box
  COERCEFLAG                 Exit, converting flag to %0 or %1
;
```

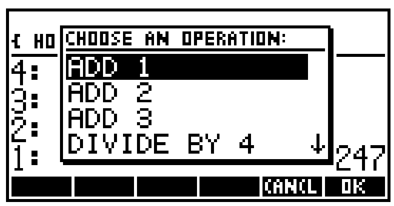


Note: You may also include the file GUI.H to enable the use of predefined decompile objects. For more about this file, see *Input Form DEFINEs for RPLCOMP* on page 187 later in this chapter.

The real power of the ability to handle lists for choices is to be able to bundle procedures with choice strings. The example CHS4 illustrates this concept.

```
CHS4 245.5 Bytes Checksum #E1FDh
( % → %' )
```

```
EXTERNAL Choose
::
  0LASTOWDOB! CK1NOLASTWD          Clear saved command name, require one ob
  CK&DISPATCH1 real                Require real number
  ::
    ' DROPFALSE                    Message handler
    "CHOOSE AN OPERATION:"         Title string
    SEVENTEEN                      Decompile object: show first part as text
    {                              Data list
      { "ADD 1" %1+ }
      { "ADD 2" :: %2 %+ ; }
      { "ADD 3" :: %3 %+ ; }
      { "DIVIDE BY 4" :: %4 %/ ; }
      { "SUBTRACT 5" :: %5 %- ; )
      { "MULTIPLY BY 6" :: %6 %* ; }
    }
  ONE                              Initial focus position
  Choose                           Run the choose box
  NOT?SEMI                         Exit if cancelled
  TWO NTHCOMPDROP                  Extract the procedure object
  EVAL                             Evaluate the procedure object
;
```



9.3.5 Customizing Choose Box Menus

By responding to message 83 you can customize the choose box menu. Rather than duplicate the definitions of the check, cancel, and OK keys, we'll illustrate how you can copy, decompose, alter, and rebuild a built-in menu definition.

There are three standard menu objects used for choose boxes:

`ChooseMenu0`	#050B3h	G/GX XLIB 179 5
Choose menu for display-only choose boxes:		
| | | | | | | | | | |QUIT|		
	→	menu_object
`ChooseMenu1`	#060B3h	G/GX XLIB 179 6
Choose menu for single-pick choose boxes:		
| | | | | | | | |CANCL| | OK |		
	→	menu_object
`ChooseMenu2`	#070B3h	G/GX XLIB 179 7
Choose menu for multi-pick choose boxes:		
| | | | | CHK | | | |CANCL| | OK |		
	→	menu_object

These menu objects are actually secondaries consisting of the object `NoExitAction` and the menu definition itself. For example, `ChooseMenu2` looks like this:

```
::
  NoExitAction
  {
    NullMenuKey
    NullMenuKey
    {
      :: TakeOver grobCheckKey ;   The grob for the label
      {
        DoCKeyCheck                Primary key checks or unchecks an item
        DoCKeyChAll                Left-shift key checks all items
        DoCKeyUnChAll              Right-shift key unchecks all items
      }
    }
    NullMenuKey
    { "(AN(L" DoCKeyCancel }
    { "OK" DoCKeyOK }
  }
;
```

(Actually, the definition for the third key is a little more involved — the check grob is not displayed if the list is empty, but if your application doesn't present an empty data list you won't have to take this step.)

The object `NoExitAction` insures that the menu won't be saved as the last menu, so pressing ⮫ [MENU] won't display a menu whose context is meaningless after your application terminates.

`NoExitAction`		#3EC58h
Ensures a menu won't be saved as the last menu		
	→	

Note: The new key definition must follow all the same principles as any key definition for the Parameterized Outer Loop (the choose box engine rests atop a POL).

Choose box menu items are built using the following support objects:

`grobCheckKey`	#860B0h	G/GX XLIB 176 134
Check label grob		
	→	grob

`DoCKeyCheck`	#2A0B3h	G/GX XLIB 179 42
Check or uncheck the current item in a multi-pick choose box		
	→	

`DoCKeyChAll`	#2B0B3h	G/GX XLIB 179 43
Check all items in a multi-pick choose box (typically left-shifted)		
	→	

`DoCKeyUnChAll`	#2C0B3h	G/GX XLIB 179 44
Uncheck all items in a multi-pick choose box (typically right-shifted)		
	→	

`DoCKeyCancel`	#2D0B3h	G/GX XLIB 179 45
Cancel the choose box		
	→	FALSE

`DoCKeyOK`	#2E0B3h	G/GX XLIB 179 46
Accept the choices		
	→	FALSE *No items chosen*
	→	*Item* TRUE *Single-pick*
	→	*Items* TRUE *Multi-pick*

Example. The technique described above is used to create a simple editor for a list of strings using a custom choose box menu. This example begins by requiring a list, validating that the list contains at least one object, and that all objects in the list are strings. The message handler for the choose box intercepts the following messages:

60 Specifies a full-screen choose box
83 Creates the custom choose box menu
96 Places the list on the stack when the choose box ends

Note that in this example we use `ONE` for the decompile object. This means we're guaranteeing to the choose box engine that only string objects are being displayed. If this example were to work with arbitrary objects, then `FOUR` would be better choice, but strings would be displayed with quote marks.

```
CHS5 320 Bytes Checksum #427h
( { $1 ... $n } → { $1 ... $n } $Highlighted %1 )   User pressed [ENTER] or | OK |
( { $1 ... $n } → %0 )                               User pressed |CANCL| or [ON]

EXTERNAL Choose
EXTERNAL DoCKeyCancel
EXTERNAL DoCKeyOK
::
  0LASTOWDOB! CK1NOLASTWD                          Clear saved command name, require one object
  CK&DISPATCH1 list                                Require list object
  ::
    DUPLENCOMP DUP#0= case SETSIZEERR              Make sure list contains at least one object
    #1+ ONE DO                                     Loop to validate objects in list
      DUP INDEX@ NTHCOMPDROP                       Get each item
      TYPECSTR? ?SKIP SETTYPEERR                   Error out if not a string
    LOOP
    ' ::                                           Message handler
      SIXTY #=casedrop :: TRUE TRUE ;              60: Full screen choose box
      83 #=casedrop                                83: Choose box menu
        ::
          ' ::                                     Place secondary on stack
          NoExitAction
          {
            {                                      Edit key definition
              "EDIT"                               Label
              :: TakeOver                          Action must begin with TakeOver
                "Edit String:"                     Set up InputLine parameters: this is the
                                                   prompt
                19GETLAM 18GETLAM                  Get the choose box data list and current item #
                NTHCOMPDROP                        Extract the highlighted item
                ZERO ONE ONE ONE                   InputLine params: alpha lock, entry, cursor
                                                   pos
{ <SkipKey >SkipKey <DelKey >DelKey TogInsertKey } Editline menu
                ONE FALSE ZERO                     Menu row, abort action, no post-processing
                InputLine                          Run the input line
                IT                                 If edit was accepted
                  ::
                    18GETLAM 19GETLAM              Get the data list and focus position
                    PUTLIST                        Replace the item
                    19PUTLAM                       Store the new list back
                  ;
                ClrDAsOK                           Signal the display has been altered
              ;                                    End of new menu key action
            }                                      End of edit key definition
            NullMenuKey                            2nd menu key
            NullMenuKey                            3rd menu key
            NullMenuKey                            4th menu key
            { "(AN(L" DoCKeyCancel }               Cancel key
            { "OK" DoCKeyOK }                      OK key
          }
          ;                                        End of menu secondary
          TRUE                                     Signal that message 83 has been handled
```

```
        ;                               End of handler for message 83
      BINT_96d #=casedrop               96: Choose box ends
        :: 19GETLAM TRUE TRUE ;         Recall data list, signal end OK, signal msg
                                        handled
      DROP FALSE                        Ignore other messages
    ;                                   End of message handler
    "EDIT STRINGS" ONE                  Choose box title, decompile specification
    4ROLL ONE                           Move data list into place, specify ONE for
                                        initial focus
    Choose                              Display the choose box
    COERCEFLAG                          Exit, converting choose box flag to %0 or %1
  ;
;
```

9.3.6 Choose Event Procedures

The following objects are available for use by a choose box menu key definition.

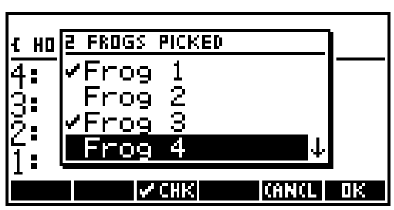
`LEDispItem`	#360B3h	G/GX XLIB 179 54
Display an item		
#index #highlight_row	→	

`LEDispList`	#350B3h	G/GX XLIB 179 53
Display the choose box contents		
	→	

`LEDispPrompt`	#300B3h	G/GX XLIB 179 48
Display the choose box title		
	→	

For `LEDispItem`, the index of the currently highlighted item can be found by `18GETLAM` and the current highlight row number can be found by `6GETLAM`.

Example. The message handler and custom menu combine in CHS6 to present a dynamic choose box in which the title reflects the number of items chosen.



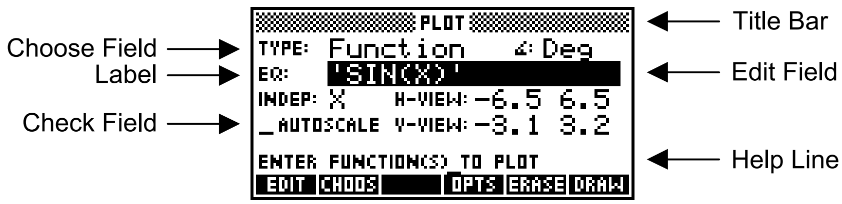

```
CHS6 348.5 Bytes Checksum #AE5Ch
( → %0 )                                User pressed |CANCL| or [ON]
( → { choices } %1 )                    User pressed [ENTER] or | OK |
```

```
EXTERNAL Choose
EXTERNAL grobCheckKey
EXTERNAL LEDispPrompt
EXTERNAL DoCKeyCheck
EXTERNAL DoCKeyChAll
EXTERNAL DoCKeyUnChAll
EXTERNAL DoCKeyCancel
EXTERNAL DoCKeyOK
::
  AtUserStack                           Clear saved command name, no arguments
  ' ::                                  Message handler
    SIXTYONE #=casedrop TrueTrue        Specify multi-pick choose box
```

```
    SIXTYTWO #=casedrop :: NINE TRUE ;                 Specify nine choices
    SEVENTY #=casedrop                                 Create the prompt string:
      ::
        15GETLAM LENCOMP                               Get the length of the list of picked indices
        ::
          ZERO #=casedrop "NO FROGS"                   No choices picked
          ONE #=casedrop "1 FROG"                      One choice picked
          UNCOERCE EDITDECOMP$ " FROGS" &$             More than one choice picked
        ;
        " PICKED" &$                                   Append remainder of prompt string
        TRUE                                           Signal event handled
      ;
    80 #=casedrop                                      Create the string for each choice:
      ::
        UNCOERCE EDITDECOMP$                           Convert index bint into real and decompile it
        "Frog " SWAP&$                                 Prepend frog string
        TRUE                                           Signal event handled
      ;
    83 #=casedrop                                      Specify the choose box menu
      ::
      ' ::
        NoExitAction
        {
          NullMenuKey
          NullMenuKey
          {
            :: TakeOver grobCheckKey ;                 Check key label
            {
              :: TakeOver DoCKeyCheck LEDispPrompt ;   Primary check key action
              :: TakeOver DoCKeyChAll LEDispPrompt ;   Left-shift key action
              :: TakeOver DoCKeyUnChAll LEDispPrompt ; Right-shift key action
            }
          }
          NullMenuKey
          { "(AN(L" DoCKeyCancel }                     Cancel key
          { "OK" DoCKeyOK }                            OK key
        }
      ;
      TRUE                                             Signal menu event handled
    ;
    DROP FALSE                                         Signal other messages not handled
  ;
  " "                                                  Default title string (will be replaced by msg 70)
  ONE                                                  Decompile object (not used in this example)
  NULL{}                                               Null data list
  ONE                                                  Initial focus position
  Choose                                               Display the choose box
  COERCEFLAG                                           Exit, converting flag
;
```

9.4 Input Forms

The input form engine in the HP 48G/GX has been designed to meet a very diverse set of requirements, so it takes a little more effort to use than other interfaces. It is not possible (or reasonable) to try to document all of the minutiae associated with input forms, but we will provide a general introduction that should satisfy the needs of many applications. We begin by introducing a few terms, then go on to describe the parameters and illustrate their use. As you read these terms, use the PLOT input form shown below for reference:



Title Bar Shows the title for the input form.

Field An input form field contains data that can be changed by the user.

Label A label is just text, and is not associated with a field except by juxtaposition.

Help Line A prompt associated with a field.

Highlight / Focus The currently active field is shown in inverse video, and is said to have the *focus* of the input form engine.

Edit Field A field that permits character editing, like the EQ field in the PLOT input form.

Choose Field A field that permits selection from a fixed set of choices, like the TYPE field in the PLOT input form.

Check Field A field that has two states: *checked* and *unchecked*, like the AUTOSCALE field in the PLOT input form.

9.4.1 Input Form Parameters

Like the choose box, input forms are specified by stack parameters and responses generated from a message handler:

`DoInputForm`		G/GX #199EBh
Display an input form		
input form parameters →	ob_1 ... ob_m TRUE	*Input accepted with OK*
input form parameters →	FALSE	*Cancelled*

Label_Specifier1 ... Label_SpecifierN Specifiers for *N* labels. Label specifiers consist of three arguments, described in detail below.

Field_Specifier1 ... Field_SpecifierM Specifiers for *M* fields. Field specifiers consist of thirteen arguments, described in detail below.

#LabelCount A binary integer *N* specifying the number of label specifiers.

#FieldCount A binary integer *M* specifying the number of field specifiers.

Input Form Message Handler A secondary that handles form-specific events.

Title A string to be displayed in the title bar.

> **Caution:** Remember that the |CALC| softkey on the second page of the input form menu gives the user access to the stack. You may wish to consider what your application leaves on the stack when an input form is active.

9.4.2 Label Specifiers

Input form labels are displayed using the small font. Each label is specified with three parameters:

Label_String A string object for the text.

#X_Position A bint specifying the pixel column for the upper-left corner of the text.

#Y_Position A bint specifying the pixel row for the upper-left corner of the text.

9.4.3 Field Specifiers

Input form fields are specified with thirteen parameters:

Field_Message_Handler A message handler, usually specified as `'DROPFALSE`.

#X_Position A bint specifying the pixel column for the upper-left corner of the field.

#Y_Position A bint specifying the pixel row for the upper-left corner of the field.

#Field_Width A bint specifying the pixel width of the field.

#Field_Height A bint specifying the pixel height of the field.

#Field_Type A bint specifying the field type. Common types are:

Value	Field Type
1	Text field
3	Auto-algebraic field for equation entry
12	Choose field
32	Check field

Object_Types A list of one or more bints specifying the valid object types for the field. To allow any object type, specify `MINUSONE`. For a check field, specify `MINUSONE`.

Decompile_Object An object specifying the manner in which the field's contents are displayed. See *Decompile Objects* on page 177 for a complete description. For a check field, specify `MINUSONE`.

Help_String A string object containing the help text for the field.

Choose_Field_Data A list of choices for a choose field, or `MINUSONE` for non-choose fields.

Choose_Decompile_Fmt An object specifying the manner in which a choose field's choices are displayed. See *Decompile Objects* on page 177 for a complete description. For non-choose fields, specify `MINUSONE`.

Reset_Value The value to be displayed if `|RESET|` is pressed. For check fields, specify `TRUE` (checked) or `FALSE` (unchecked). For other fields, specify `MINUSONE` if the reset value for the field is blank (analogous to `NOVAL` in User-RPL) or specify a valid value.

Initial_Value The first value to be displayed. For check fields, specify `TRUE` (checked) or `FALSE` (unchecked). For other fields, specify `MINUSONE` if the reset value for the field is blank (analogous to `NOVAL` in User-RPL) or specify a valid value.

Looks easy, right? Let's put the first example right on the next page:

```
INF1 287 Bytes Checksum #D6D6h
( → %0 )                          Cancelled
( → ob % % %1 )                   Accepted
::
AtUserStack                       Clear saved command name, no arguments
"EDIT FIELD:" ONE NINETEEN        Label 1 text and coordinates
"CHOOSE FIELD:" ONE TWENTYEIGHT   Label 2 text and coordinates
"CHECK FIELD" EIGHT THIRTYSEVEN   Label 3 text and coordinates

'DROPFALSE                        Field 1 message handler
FORTY SEVENTEEN                   Field 1 coordinates
79                                Field 1 width
NINE                              Field 1 height
ONE                               Field 1 type — edit field
MINUSONE                          Field 1 object types allowed
TWO                               Field 1 decompile format user's settings
"ENTER ANY OBJECT"                Field 1 help text
MINUSONE                          Optional data not used
MINUSONE                          Optional data not used
NULL$ NULL$                       Field 1 initial and reset values

'DROPFALSE                        Field 2 message handler
FORTYNINE TWENTYSIX               Field 2 coordinates
FORTYNINE                         Field 2 width
NINE                              Field 2 height
TWELVE                            Field 2 type — choose list
FOUR                              Field 2 object types allowed
TWO                               Field 2 decompile format user's settings
"CHOOSE A NUMBER"                 Field 2 help text
{ %1 %2 %3 )                      Field 2 choice list
TWO                               Choose box decompile format
%1 %1                             Field 2 initial and reset values

'DROPFALSE                        Field 3 message handler
ONE THIRTYFIVE                    Field 3 coordinates
SIX                               Field 3 width
NINE                              Field 3 height
THIRTYTWO                         Field 3 type — check box
MINUSONE                          Object types not applicable
MINUSONE                          Decompile format not applicable
"CHECK OR UNCHECK"                Field 3 help text
MINUSONE                          Optional data not used
MINUSONE                          Optional data not used
FALSE FALSE                       Field 3 initial and reset values

THREE                             Number of labels
THREE                             Number of fields
'DROPFALSE                        input form message handler
"TEST"                            input form title
DoInputForm                       Display the input form
case :: ITE %1 %0 %1 ;            If OK, convert check result and return %1
%0                                If cancelled, return %0
;
```

9.4.4 Input Form DEFINEs for RPLCOMP

The example INF1 on the previous page is virtually unreadable unless you're willing to remember many small details of input form parameters. To solve this, you can use the INCLUDE feature of HP's RPL compiler RPLCOMP.EXE to define locations for fields and labels, field types, decompile procedures, etc. We've provided a file on the disk named `GUI.H` that contains some standard input form definitions. If you're using another tool set, there may be a similar way to use DEFINEs to help make your code readable.

Note: The remaining examples in this chapter will use the DEFINEs listed in GUI.H.

Example. INF2 is slightly different from INF1. The first two fields are lined up to begin in the same pixel column, the decompile specifications use STD instead of the user settings, and `NOVAL` is the default for field 1. We trust that the mnemonic value of the DEFINEs from GUI.H makes the code a little more readable.

```
INF2 287 Bytes Checksum #3373h
( → %0 )                        Cancelled
( → ob % % %1 )                 Accepted
INCLUDE GUI.H                   Include the DEFINEs from file GUI.H
::
  AtUserStack                   Clear saved command name, no arguments
  "EDIT FIELD:" COL1 LROW2      Label 1 text and coordinates
  "CHOOSE FIELD:" COL1 LROW3    Label 2 text and coordinates
  "CHECK FIELD" COL1+C LROW4    Label 3 text and coordinates

  'DROPFALSE                    Field 1 message handler
  COL9 FROW2 FWIDTH12 FHEIGHT   Field 1 coordinates and dimensions
  FTYPE_TEXT                    Field 1 type: edit field
  OBTYPE_ANY                    Field 1 object types allowed
  FMT_STD                       Field 1 decompile format STD
  "ENTER ANY OBJECT"            Field 1 help text
  OPTDATA_NULL                  Optional data not used
  OPTDATA_NULL                  Optional data not used
  NOVAL NOVAL                   Field 1 initial and reset values

  'DROPFALSE                    Field 2 message handler
  COL9 FROW3 FWIDTH8 FHEIGHT    Field 2 coordinates and dimensions
  FTYPE_CHOOSE                  Field 2 type: choose list
  OBTYPE_NA                     Field 2 object types allowed
  FMT_STD                       Field 2 decompile format STD
  "CHOOSE A NUMBER"             Field 2 help text
  { %1 %2 %3 }                  Field 2 choice list
  FMT_STD                       Choose box decompile format
  %1 %1                         Field 2 initial and reset values

  'DROPFALSE                    Field 3 message handler
  COL1 FROW4 FWIDTH_C FHEIGHT   Field 3 coordinates and dimensions
  FTYPE_CHECK                   Field 3 type: check box
  OBTYPE_NA                     Object types not applicable
  FMT_NA                        Decompile format not applicable
  "CHECK OR UNCHECK"            Field 3 help text
  OPTDATA_NULL                  Optional data not used
  OPTDATA_NULL                  Optional data not used
  FALSE FALSE                   Field 3 initial and reset values

  THREE THREE                   Number of labels and fields
  'DROPFALSE                    Input form message handler
  "TEST"                        Input form title
  DoInputForm                   Display the input form
  case :: ITE %1 %0 %1 ;        If OK, convert check result and return %1
  %0                            If cancelled, return %0
;
```

9.4.5 Specifying Object Types

To allow any object to be entered into a text field, specify `MINUSONE` for the object type. To specify one or more object types, use a list of bints. The table below shows the available types, bint values, and DEFINE names from `GUI.H`.

Object Type	DEFINE	Bint
Real	OBTYPE_REAL	ZERO
Complex	OBTYPE_CMP	ONE
String	OBTYPE_STR	TWO
Real array	OBTYPE_RARRAY	THREE
Complex array	OBTYPE_CARRAY	FOUR
List	OBTYPE_LIST	FIVE
Name (ID)	OBTYPE_ID	SIX
User program	OBTYPE_USERPRGM	EIGHT
Algebraic	OBTYPE_SYMB	NINE
User binary integer	OBTYPE_HXS	TEN
Unit	OBTYPE_UNIT	THIRTEEN

Example: To allow programs and algebraic objects use the list `{ OBTYPE_USERPRGM OBTYPE_SYMB }`.

9.4.6 Specifying Decompile Formats

Text and choose fields require a decompile object. The decompile object controls the manner in which each item is displayed, has the stack diagram (ob → $), and may be specified three ways:

- A pointer to an object that creates a string representation of a choice, like `EDITDECOMP$`
- A secondary that creates a string representation of a choice, like `:: CARCOMP EDITDECOMP$ ;`
- A bint specifying the decompile procedure

Note that for text fields, the first two choices must be sensitive to the possibility of undefined field contents. For instance, if a text field's default value is `MINUSONE` (NOVAL), then `EDITDECOMP$` would display `<FFFFFh>`. It's more likely that a secondary would be used that would include a test for this condition.

Example: This secondary returns a null string for an undefined value, otherwise decompiles the object using STD formatting if the object is not a string.

```
( ob → $ )
::
  DUP MINUSONE EQUAL casedrop NULL$   Return null string for NOVAL
  DUPTYPECSTR? ?SEMI                  Do nothing if the object is a string
  EDITDECOMP$
;
```

The binary integer specification uses specific bits to encode the decompile procedure. These bits control the decompile format, which part of a composite choice to decompile, and whether only the first character should be returned. The file `GUI.H` contains a series of DEFINEs for commonly used decompile specifications.

Bit	Interpretation
0	No decompilation — expects a string and displays the contents without quote marks
1	Decompile objects as they would appear on the stack (uses the user's numeric display format settings)
2	Decompile objects as they would appear in the editline (uses STD format for numbers)
3	Return only the first character of the string
4	Extract and display the first object of a composite (useful for choose fields only)
5	Extract and display the second object of a composite (useful for choose fields only)

Example: The bint `THIRTYSIX` (`FMT_P2&STD` in GUI.H) specifies STD formatting for the second element in a list (useful for choose fields).

9.4.7 Input Form Message Handlers

At various times during the execution of an input form, the input form engine sends a message to the form's message handler or an individual field's message handler. If the message handler chooses not to handle the message, the default behavior related to that message will occur. If the message handler does handle the message, the default behavior does not happen. If you don't plan to handle any messages, then the object `DROPFALSE` is all that's needed.

A message arrives at the message handler in the form of a binary integer indicating the message type with optional stack parameters. The message handler is expected to return TRUE if the message was handled, along with any required results on the stack, or FALSE if the message was not handled.

A message handler has the following stack diagram:

<passed objects> #message → <*returned objects*> TRUE
<passed objects> #message → <passed objects> FALSE

There are many messages, but the messages most likely to be of interest are documented as follows:

Message Purpose			Decimal message number
	Input arguments	→	Objects returned by the handler

Input Form Messages

These messages are processed by the main input form message handler.

Message Purpose	Input arguments		Objects returned	Number
`Title Grob`				2
		→	131x7_grob	
`Input Form Menu`				15
		→	{ menu }	
`Three Menu Keys`				16
		→	{ Key_4 Key_5 Key_6 }	
|CALC| `Key Event`				28
		→	FALSE *Cancel not allowed*	
		→	TRUE *Cancel allowed*	
| OK | `Key Event`				29
		→	FALSE *OK not allowed*	
		→	TRUE *OK allowed*	

Field Messages

These messages are processed by the individual field message handlers and are specific to the related field.

Message Purpose	Input arguments		Objects returned	Number
`Check Object Type`				45
		→	FALSE *Invalid Object Type*	
		→	TRUE *Valid Object Type*	
`Check Object Value`				46
		→	FALSE *Invalid Object Value*	
		→	TRUE *Valid Object Value*	

9.4.8 Input Form Data Access

While an input form is active the objects `gFldVal` and `GetFieldVals` may be used to recall the values for all the fields. Fields are numbered in the order of their specification.

`gFldVal`	#C50B0h	G/GX XLIB 176 197
Recall the values for an individual field		
#field_number	→	*Field_Value*
`GetFieldVals`	#C80B0h	G/GX XLIB 176 200
Recall the values for all the fields		
	→	*Field_Values*

Example: `:: ONE gFldVal ;` returns the value of the first field.

While an input form is active, state information is saved in null-named temporary variables. A few contain basic information that might be useful:

4GETLAM	→	#current_field_number
5GETLAM	→	#focus_position
12GETLAM	→	$title
14GETLAM	→	#number_of_fields
15GETLAM	→	#number_of_labels

9.4.9 Customizing Input Form Menus

There are twelve standard input form softkeys:

	Key 1	Key 2	Key 3	Key 4	Key 5	Key 6
Row 1	|EDIT |	|CHOOS|	| CHK |	| |	|CANCL|	| OK |
Row 2	|RESET|	|CALC |	|TYPES|	| |	|CANCL|	| OK |

In row 1, the first three keys are reserved for field support. The last three are available for customization by responding to message 16. If an application doesn't need the second row (the |CALC| key represents a potential landmine for a robust application), the entire menu can be customized by responding to message 15.

Two built-in key objects are available to help build custom input form menus: `DoKeyCancel` and `DoKeyOK`:

`DoKeyCancel`	#590B0h	G/GX XLIB 176 89
Process a "CANCEL" keystroke, terminating an input form		
	→	*FALSE*
`DoKeyOK`	#5A0B0h	G/GX XLIB 176 90
Process an "OK" keystroke, terminating an input form		
	→	*Field_Values TRUE*

Customizing Three Menu Keys. By responding to message 16, you can supply your own keys for row 1 positions four, five, and six. You must supply a list of exactly three key definitions and TRUE (in addition to the TRUE indicating that the message has been handled).

The following input form message handler creates a new key |ALERT| in position four and supplies the standard |CANCL| and | OK | keys in positions five and six:

(#msg → FALSE *Not handled*)
(#16 → { Key_1 Key_2 Key_3 } TRUE TRUE)

```
::
  SIXTEEN #<> case FALSE                    Respond only to message 16
  {                                         List of 3 key definitions:
    {                                       Key 1:
      "ALERT"                               Label
      ::                                    Procedure:
        TakeOver                              MUST be a TakeOver secondary
        "Alert!"                              Text for message box
        NINE FIFTEEN                          Min and max character widths
        MINUSONE                              No grob
        ' MsgBoxMenu                          Message box menu
        DoMsgBox                              Display the message box
        DROP                                  Discard the returned flag
      ;
    }
    { "(AN(L" :: TakeOver DoKeyCancel ; }   Standard |CANCL| key
    { "OK" :: TakeOver DoKeyOK ; }          Standard | OK | key
  }
  TRUE                                      Flag needed by menu builder
  TRUE                                      Indicates message handled
;
```

The program INF3 (supplied on the disk but not listed here) uses this message handler to extend the INF2 example.

Customizing the Entire Input Form Menu. There are two principal motivations for customizing the entire input form menu:

- You can rename a standard key, like | OK | to a verb, like |DRAW| in the PLOT input form.
- You can eliminate keys that are either distracting or dangerous. Keys like |RESET| and |TYPES| are distracting in a well-confined application, but |CALC| is quite dangerous, since this key gives the user access to the entire calculator.

By responding to message 15, you can supply a unique menu definition. The menu definition must be supplied as a secondary consisting of two parts — `NoExitAction` and the menu list:

```
:: NoExitAction { menu keys } ;
```

To help build the menu, you can use the standard first three keys that are available in the list `IFMenuRow1`, and the standard second menu row which is available in the list `IFMenuRow2`.

`IFMenuRow1`	#050B0h	G/GX XLIB 176 5
A list containing the standard first three input form softkeys → { *EDIT CHOOSE CHK* }		
`IFMenuRow2`	#060B0h	G/GX XLIB 176 6
A list containing the standard second row of input form softkeys → { *RESET CALC TYPES NullMenuKey CANCEL OK* }		

The following input form message handler creates a new key |ALERT| in position four and supplies the standard |CANCL| and | OK | keys in positions five and six:

```
( #msg → FALSE Not handled )
( #16 → { Key1 Key2 Key3 } TRUE TRUE )
::
  FIFTEEN #<> case FALSE               Respond only to message 15
  ' NoExitAction                       Place NoExitAction on the stack
  IFMenuRow1                           Get the first three standard keys
  {                                    List of 3 key definitions:
    {                                  Key 1:
      "ALERT"                          Label
      ::                               Procedure:
        TakeOver                         MUST be a TakeOver secondary
        "Alert!"                         Text for message box
        NINE FIFTEEN                     Min and max character widths
        MINUSONE                         No grob
        ' MsgBoxMenu                     Message box menu
        DoMsgBox                         Display the message box
        DROP                             Discard the returned flag
      ;
    }
    { "(AN(L" TakeOver DoKeyCancel ; }  Standard |CANCL| key
    { "OK" TakeOver DoKeyOK ; }         Standard | OK | key
  }
  &COMP                                Concatenate the two lists
  TWO ::N                              Build the secondary
  TRUE                                 Indicates message handled
;
```

The program INF4 (supplied on the disk but not listed here) uses this message handler to extend the INF3 example. Note that INF3 and INF4 are identical *except* that INF4 does not have the second row of standard input form keys.

9.4.10 ORBIT Example

This program is a System-RPL implementation of an example by the same name in *The HP48 Handbook* (also provided on the disk in the USERRPL directory). ORBIT models a particle in a chaotic orbit. This program was inspired by the program MIRA in the book *Fractals — Endlessly Repeated Geometrical Figures* (Princeton, New Jersey: Princeton University Press, 1991) by Hans Lauwerier.

The successive iterates are calculated by:

$$x_{n+1} = y_n - F(x_n)$$

$$y_{n+1} = -bx_n + F(x_{n+1})$$

where:

$$F(x) = ax + \frac{2(1-a)x^2}{1+x^2}$$

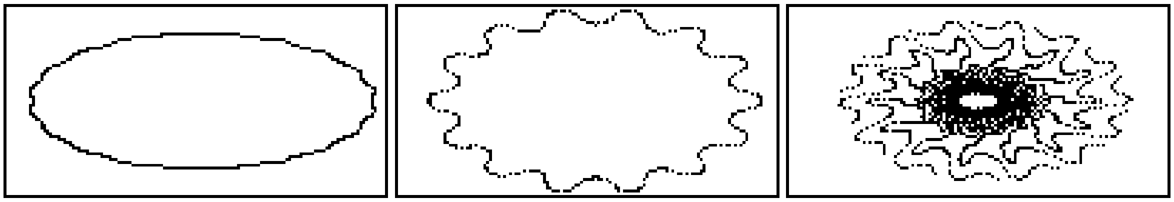
The value for a controls the chaotic behavior (orbits are stable when a is 1). The value of b controls the spiral nature of the orbit. If b is just slightly less than 1, the orbit spirals inward.

An input form is used to enter and verify the input parameters n (the number of iterates), initial values for a and b, the starting position x and y, and the scaling coordinates. There are two message handlers:

- The field message handler for n verifies a positive number of iterates.
- The form message handler provides a custom menu that adds a |SHOW|, renames | OK | to |DRAW|, verifies that all fields have data when |DRAW| is pressed, and omits the standard second menu row.

To get acquainted with ORBIT, begin with a somewhat stable orbit. Reduce a to see its effect on the orbit and adjust the scale to keep the picture large, then reduce b to make the orbit spiral inward:

n	a	b	x	y	PMIN	PMAX
700	.95	1	0	7.5	(-25,-10)	(27,10)
700	.9	1	0	7.5	(-20,-8)	(22,8)
2200	.9	.998	0	7.5	(-20,-8)	(22,8)



Here's some more to try. Remember that very small variations in initial conditions can result in dramatic changes to the orbit. For instance, try the third example below with values for a of -.24, -.25, and -.26.

n	a	b	x	y	PMIN	PMAX
600	-.4	.99	4	0	(-12,-10)	(13,10)
900	-.48	.935	4.1	0	(-11,-10)	(14,7)
500	-.05	.985	9.8	0	(-13,-11)	(17,11)
1000	-.24	.998	3	0	(-12,-10)	(14,10)
1000	.2	1	11	0	(-20,-16)	(22,17)
400	.3	1	8	0	(-35,-19)	(35,19)
500	.4	1	0	5	(-13,-8)	(16,8)

```
ORBIT 1278.5 Bytes Checksum #E440h
( → )
```

```
INCLUDE GUI.H                              Include input form DEFINEs

EXTERNAL DoKeyCancel                       External declarations for objects that are
EXTERNAL DoKeyOK                           referenced by rompointer
EXTERNAL IFMenuRow1
EXTERNAL gFldVal
EXTERNAL GetFieldVals
EXTERNAL grobAlertIcon
EXTERNAL DoMsgBox
EXTERNAL MsgBoxMenu

::
  AtUserStack                              No arguments, clear saved command name

Specify the input form labels:

  "ITERATES:" COL1 LROW1                   input form labels
  "A:" COL1 LROW2
  "B:" COL12 LROW2
  "X:" COL1 LROW3
  "Y:" COL12 LROW3
  "PMIN:" COL1 LROW4
  "PMAX:" COL12 LROW4

Specify the input form fields:
```

```
' ::                                           Message handler for ITERATES field
  FORTYSIX #<> case FALSE                      Respond only to message 46
  %0 %>                                        Test to see if number is greater than zero
  TRUE                                         Signal that the message has been handled
;
COL7 FROW1 FWIDTH8 FHEIGHT                     Field dimensions
FTYPE_TEXT                                     Field type
{ OBTYPE_REAL }                                Allow only real numbers
' ::                                           Decompile object
  DUP MINUSONE EQUAL casedrop NULL$            Show null string if no data has been entered
  EDITDECOMP$                                  Else display in STD format (similar to
                                               FMT_STD)
;
"ENTER THE NUMBER OF ITERATES"                 Help text
OPTDATA_NULL OPTDATA_NULL                      No choose box data for a text field
NOVAL NOVAL                                    No value for reset and initial values

'DROPFALSE                                     Default message handler for A field
COL2 FROW2 FWIDTH8 FHEIGHT                     Field dimensions
FTYPE_TEXT                                     Field type
{ OBTYPE_REAL }                                Allow only real numbers
FMT_STD                                        Use STD display formatting
"'A' CONTROLS THE CAOTIC BEHAVIOR"             Help text
OPTDATA_NULL OPTDATA_NULL                      No choose box data for a text field
NOVAL NOVAL                                    No value for reset and initial values

'DROPFALSE                                     Default message handler for B field
COL13 FROW2 FWIDTH8 FHEIGHT                    Field dimensions
FTYPE_TEXT                                     Field type
{ OBTYPE_REAL }                                Allow only real numbers
FMT_STD                                        Use STD display formatting
"'B' CONTROLS THE SPIRAL"                      Help text
OPTDATA_NULL OPTDATA_NULL                      No choose box data for a text field
NOVAL NOVAL                                    No value for reset and initial values

'DROPFALSE                                     Default message handler for X field
COL2 FROW3 FWIDTH8 FHEIGHT                     Field dimensions
FTYPE_TEXT                                     Field type
{ OBTYPE_REAL }                                Allow only real numbers
FMT_STD                                        Use STD display formatting
"'X' IS THE STARTING POSITION X"               Help text
OPTDATA_NULL OPTDATA_NULL                      No choose box data for a text field
NOVAL NOVAL                                    No value for reset and initial values

'DROPFALSE                                     Default message handler for Y field
COL13 FROW3 FWIDTH8 FHEIGHT                    Field dimensions
FTYPE_TEXT                                     Field type
{ OBTYPE_REAL }                                Allow only real numbers
FMT_STD                                        Use STD display formatting
"'Y' IS THE STARTING POSITION Y"               Help text
OPTDATA_NULL OPTDATA_NULL                      No choose box data for a text field
NOVAL NOVAL                                    No value for reset and initial values

'DROPFALSE                                     Default message handler for PMIN
COL4.5 FROW4 FWIDTH7 FHEIGHT                   Field dimensions
FTYPE_TEXT                                     Field type
{ OBTYPE_CMP }                                 Allow only complex numbers
```

```
  FMT_STD                                     Use STD display formatting
  "LOWER LEFT DISPLAY COORDINATE"             Help text
  OPTDATA_NULL OPTDATA_NULL                   No choose box data for a text field
  NOVAL NOVAL                                 No value for reset and initial values

  'DROPFALSE                                  Default message handler for PMAX
  COL15.5 FROW4 FWIDTH7 FHEIGHT               Field dimensions
  FTYPE_TEXT                                  Field type
  { OBTYPE_CMP }                              Allow only complex numbers
  FMT_STD                                     Use STD display formatting
  "UPPER RIGHT DISPLAY COORDINATE"            Help text
  OPTDATA_NULL OPTDATA_NULL                   No choose box data for a text field
  NOVAL NOVAL                                 No value for reset and initial values
```

Now specify the remaining input form parameters

```
  SEVEN                                       Seven labels
  SEVEN                                       Seven fields
  ' ::                                        Message handler:
    FIFTEEN #=casedrop                        Message 15: input form menu
    ::
      ' NoExitAction                          Put NoExitAction on the stack
      IFMenuRow1                              List of first three standard keys
      {                                       List of last three custom keys:
        {
          "SHOW"                              Label for SHOW key
          ::
            TakeOver                          Must be a TakeOver secondary
            DOCLLCD                           Clear the display
            TURNMENUOFF                       Turn off the menu
            5GETLAM gFldVal                   Get the value for the current field
            DUP MINUSONE EQUAL                Test to see if the field is undefined
            ITE                               If undefined,
              :: DROP "Undefined" ;             display "Undefined"
              EDITDECOMP$                       else decompile the value
            DISPROW4                          Display the string
            "Press any key to continue\1F"
            $>grob                            Build the prompt grob
            HARDBUFF ZERO FIFTYSIX GROB!      Display the prompt grob
            WaitForKey 2DROP                  Wait for a key, discard the location
            TURNMENUON                        Turn the menu back on
          ;
        }
        {
          "(AN(L"                             Standard CANCEL key
          :: TakeOver DoKeyCancel ;
        }
        {
          "DRAW"                              Standard OK key with different label
          :: TakeOver DoKeyOK ;
        }
      }
      &COMP                                   Concatenate the two lists of key definitions
      TWO ::N                                 Build the secondary with NoExitAction
      TRUE                                    Signal the message was handled
    ;
    TWENTYNINE #<> case FALSE                 Reject all messages other than 29
```

```
    GetFieldVals                              Get the field values
    15GETLAM                                  Get the number of field values
    TRUE 1LAMBIND                             Bind TRUE in a temporary variable
    ZERO_DO (DO)                              Loop to test each value
      MINUSONE EQUAL IT :: FALSE 1PUTLAM ;    If a value is undefined, store FALSE in temp
                                              var
    LOOP
    1GETABND                                  Recall flag, abandon temporary environment
    DUP ?SKIP                                 If there was an undefined value
      ::
        "Undefined\0AValue"                   Display a message box
        NINE FIFTEEN
        grobAlertIcon
        MsgBoxMenu
        DoMsgBox
        DROP
      ;
    TRUE                                      Signal that message 29 was handled
  ;
  "ORBIT"                                     Title for the input form
```

Now display the input form

```
  DoInputForm                                 Display the input form
  NOT?SEMI                                    Quit if cancelled
```

The user pressed DRAW, the parameters were verified, and now we're ready to go. The stack at this point contains:

(#Iterates %a %b %x %y C%PMIN C%PMAX →)

```
  C%>% PUTYMAX PUTXMAX                        Store PMIN
  C%>% PUTYMIN PUTXMIN                        Store PMAX
  BINT_131d SIXTYFOUR MAKEPICT#               Create blank PICT
  TOGDISP ZEROZERO WINDOWXY TURNMENUOFF       Display PICT with no menu
  %2 5PICK %2 %* %-                           Calculate intermediate value
  3PICK DUP %* DUP                            Calculate initial value for w
  3PICK %*
  7PICK 6PICK %* %+
  SWAP %1 %+ %/
  %0                                          Initial value for z
  { LAM a LAM b LAM x LAM y LAM c LAM w LAM z )
  BIND                                        Create local variables
  COERCE ZERO DO                              Loop for n iterations
    ATTN? IT ZEROISTOPSTO                     Quit if ATTN pressed
    LAM x INDEX@ TEN #> IT                    Plot only after 1st 10 points
      :: DUP LAM y %>C% C%># PIXON3 ;
    ' LAM z STO                               Save old x in z
    LAM b LAM y %* LAM w %+                   Calculate new x
    DUP ' LAM x STO
    LAM a OVER %* SWAP DUP %*                 Calculate new w
    DUP LAM c %* SWAP %1 %+ %/ %+
    DUP ' LAM w STO
    LAM z %- ' LAM y STO                      Complete new value for y
  LOOP
  ABND                                        Abandon temporary environment when done
  ATTNFLGCLR FLUSHKEYS                        Clear the attention flag and flush the key buffer
;
```

Chapter 10

Introducing Saturn

There are times in application development when System-RPL simply won't do the job or is too inefficient, so you want to write some code in assembly language. We summarize the CPU and instruction set here, but we also encourage you to review the document SASM.DOC supplied by Hewlett-Packard (on the disk). In particular, SASM.DOC provides some detailed information about each instruction (opcode, cycles to execute, etc.) that we omit here.

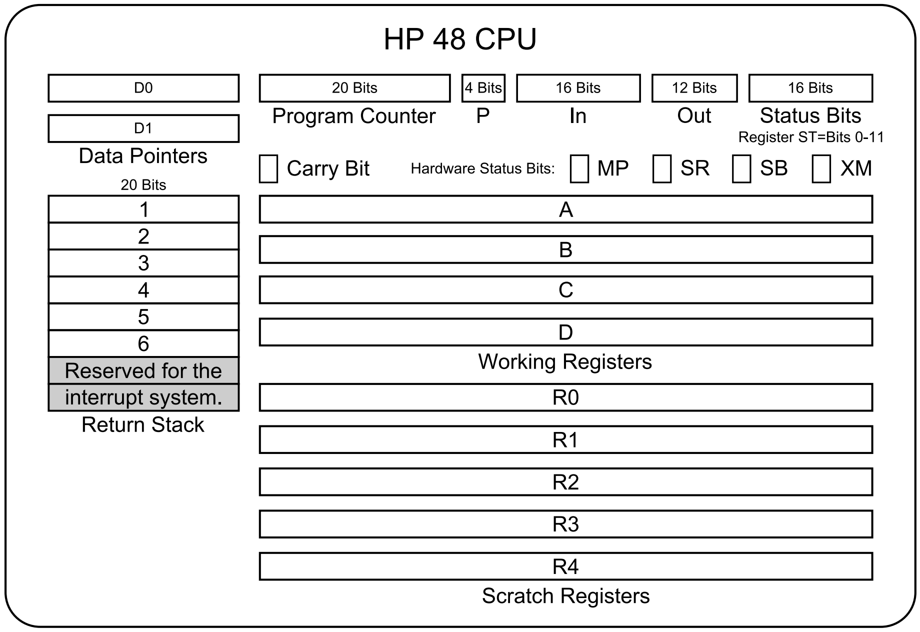
Hewlett-Packard has used the Saturn CPU since the early 1980s for the core of all calculators and the HP-71B handheld BASIC computer. Several variations of ICs using this CPU have evolved over the years, but the chip used in the HP 48 family represents the most mature implementation. The CPU is optimized for BCD math and low power consumption, traits which have helped characterize HP calculators for many years.

We begin by introducing the CPU, the instruction set. The basic mechanics of the RPL/assembler interface from the programmer's perspective are then introduced in the next chapter.

The Saturn architecture is based on a 4-bit bus, thus data is accessed a half byte at a time (these quantities are called "nibbles"). The physical address space is 512K bytes — addresses are represented as 20-bit quantities. Programs written in assembly language should be written so as to be completely relocatable in the address space.

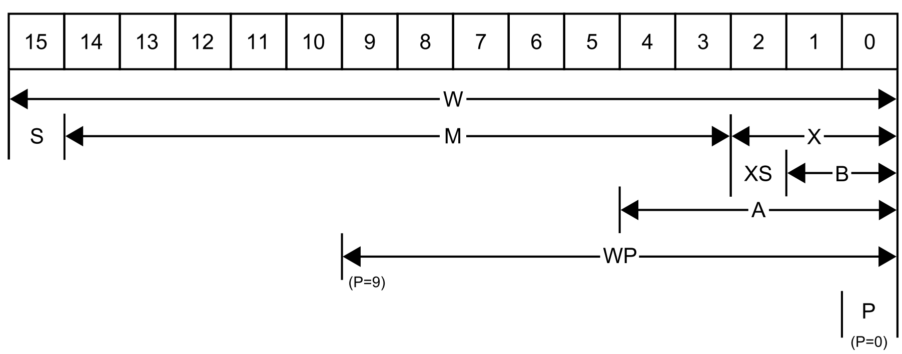
10.1 The Saturn CPU

The CPU has four working registers (A–D) and five scratch registers (R0–R4), each 64 bits wide. The data pointer registers, program counter, and return stack are all 20 bits wide. A four-bit pointer register P is used to point into the working registers. The input register is 16 bits wide, and the output register is 12 bits wide. The low-order 12 status bits are called register ST.



10.1.1 The Working and Scratch Registers

The working registers A–D, the pointer register P, and the scratch registers are the workbench of the CPU. The 64-bit (16-nibble) working registers A–D are used for data manipulation, and are divided into 9 *fields* as follows:



Field	Description
W	Word (all 16 nibbles)
A	Address field (nibbles 0–4)
B	Byte (nibbles 0 & 1)
X	Exponent (nibbles 0–2)
XS	Exponent sign (nibble 2)
M	Mantissa (nibbles 3–14)
S	Mantissa sign
P	Nibble referenced by the P register
WP	Nibbles 0 — the nibble referenced by the P register

As mentioned earlier, the CPU has been optimized for BCD math, and the fields S, M, XS, and X are commonly used in BCD math routines. The A field is most frequently used for address and object size calculations.

The A and C registers are used for memory access via the data pointers and can also exchange data with the five 64- bit scratch registers. Instructions like `A=R0` move the entire contents of R0 into A, but instructions like

`R0=A.F X` permit field specific data exchange between working and scratch registers. In the latter example, the X field of register R0 gets the contents of the X field of register A.

A note about notation: sometimes we refer to a specific field in a specific register by enclosing the field in brackets. For instance, C[A] refers to the A field of the C register.

10.1.2 The Status Bits

Carry. The carry bit is affected by calculation or logical test operations.

Carry is set if:

- A register or data pointer is incremented and overflows
- A register or data pointer is decremented and underflows
- An add operation overflows
- A subtract operation borrows
- A test is true

Carry is cleared if:

- A register or data pointer is incremented and does not overflow
- A register or data pointer is decremented and does not underflow
- An add operation does not overflow
- A subtract operation does not borrow
- A test is false

Status Bits. There are 16 status bits referred to collectively as "status bits" (not to be confused with *hardware status bits*). The lower 12 bits compose register ST. Information in register ST can be swapped with the X field of the C register. The upper four bits are reserved for use by the operating system, but for most applications the lower 12 are available.

Bit	Name
12	Deep Sleep override
13	Indicates interrupt service occurred
14	Indicates interrupt system active
15	Disable interrupts

Hardware Status Bits. The hardware status bits are:

Bit	Symbol	Name
0	XM	External Module Missing
1	SB	Sticky Bit
2	SR	Service Request
3	MP	Module Pulled

The Sticky Bit (SB) is the only one of these of interest to programmers writing applications for the HP 48. This bit is set when a non-zero bit is shifted off the right end (least significant) of a register. SB is only cleared by a `SB=0` instruction. There is a `?SB=0` instruction to test if the Sticky Bit is zero, but there is *not* a corresponding ?SB=1 test to see if the SB is set.

10.1.3 Input and Output Registers

The 16-bit input (IN) register and the 12-bit output (OUT) register are used to exchange data with the system bus. They will be used for key scanning in an example shown later. Key scanning and sound effects are the only uses you'll likely have for these registers when writing code objects for the HP 48.

10.1.4 The Return Stack

Note that two levels of the hardware return stack are reserved for the interrupt system — applications should *never* use more than 6 levels of the return stack.

10.1.5 Arithmetic Mode

The Saturn CPU can perform register arithmetic in either hexadecimal (HEX) or decimal (DEC) modes. The default mode for most operations in the HP 48 is HEX mode, however the math routines frequently use DEC mode. The instructions `SETHEX` and `SETDEC` set these modes. If you write a code object that uses DEC mode, be certain to execute `SETHEX` before returning to RPL, otherwise the HP 48 will crash. There are no test instructions or status bits for the arithmetic mode, but the two instructions

```
LCHEX 9
C=C+1 P
```

or

```
LAHEX 9
A=A+1 P
```

will set the carry bit if the CPU is in decimal mode.

Instructions which increment or decrement P, D0, or D1 are always performed in HEX mode. Also, instructions which add or subtract a constant from a specific field will be performed in HEX mode.

10.1.6 The Pointer Register

The pointer register P is a four-bit register used in field selections with the working registers. The pointer register is also useful as a tiny counter register. P may be set, incremented, decremented, or exchanged with the C register.

10.2 Instruction Set Summary

The following instruction section summarizes the Saturn instruction set. For detailed information about each instruction, see the HP document SASM.DOC.

The SASM assembler defines four fields for each instruction which contain an optional *label*, an *opcode*, the optional *modifier*, and optional *comments*: Standard practice for SASM usage is for the opcode field to begin in column 9, the modifier field to begin in column 17, and comments to begin in column 33:

Columns:	1	9	17	33
Fields:	label	opcode	modifier	Comments
Example:	`NextLevel`	`D1=D1+`	`5`	`Point D1 to next stack level`

Any source code line beginning with * will be treated as a comment.

10.2.1 Memory Access Instructions

Data Pointer Instructions.

In the following instructions,

- *r* = A or C
- *ss* = D0 or D1
- *n* is an expression whose hex value is from 0 through F
- *nnnnn* is an expression whose hex value is from 0 through FFFFF

During those operations that involve a calculation, the carry flag is set if the calculation overflows or borrows, otherwise the carry flag is cleared.

Instruction	Description	Examples
*rss*EX	Exchange A field in *r* with *ss*	`AD0EX`
*rss*XS	Exchange nibbles 0 through 3 with *ss*	`AD0XS`
ss=*r*	Copy A field in *r* into *ss*	`D1=C`
ss=*r*S	Copy nibbles 0 through 3 in *r* into *ss*	`D1=AS`
ss=*ss*+ *n*	Increment *ss* by *n*	`D1=D1+ 5`
ss=*ss*- *n*	Decrement *ss* by *n*	`D0=D0- 16`
ss=(2) *nnnnn*	Load *ss* with two nibbles from *nnnnn*	`D0=(2) A3`
ss=(4) *nnnnn*	Load *ss* with four nibbles from *nnnnn*	`D0=(4) FFC7`
ss=(5) *nnnnn*	Load *ss* with *nnnnn*	`D0=(5) =DSKTOP`

Data Transfer Instructions.

In the following instructions,

- *r* = A or C
- *fs* = A, P, WP, XS, X, S, M, B, W, or a number *n* from 1 through 16

Instruction	Description	Examples
r=DAT 0 *fs*	Copy data at address contained in D0 into *fs* field in *r* (or nibble 0 through nibble *n*-1 in *r*)	`C=DAT0 A` `A=DAT0 5`
r=DAT1 *fs*	Copy data at address contained in D1 into *fs* field in *r* (or nibble 0 through nibble *n*-1 in *r*)	`C=DAT1 B` `A=DAT1 1`
DAT0=*r* *fs*	Copy data of *fs* field in *r* (or in nibble 0 through nibble *n*-1 in *r*) to address contained in D0	`DAT0=C A` `DAT0=A 3`
DAT1=*r* *fs*	Copy data of *fs* field in *r* (or in nibble 0 through nibble *n*-1 in *r*) to address contained in D1	`DAT1=C A` `DAT1=A 3`

10.2.2 Load Constant Instructions

In the following instructions,

- *h* is a hex digit
- *i* is an integer from 1 through 5
- *nnnnn* is an expression with hex value from 0 through FFFFF
- *c* is an ASCII character

During a load constant operation, the nibbles are loaded beginning at r(P), least significant nibble first. Load operations can wrap from r(15) to r(0). A common coding mistake is to forget the setting of P during a load constant operation.

Instruction	Description	Examples
`LAHEX` *h ... h*	Load up to 16 hex digits into A.	`LCASC F247`
`LA(`*i*`)` *nnnnn*	Load *i* hex digits from the value of *nnnnn* into A.	`LAHEX 4142`
`LAASC '`*c ... c*`'`	Load up to eight ASCII characters into A.	`LAHEX 'AB'`
`LCHEX` *h ... h*	Load up to 16 hex digits into C.	`LAASC F247`
`LC(`*i*`)` *nnnnn*	Load *i* hex digits from the value of *nnnnn* into C.	`LCHEX 4142`
`LCASC '`*c ... c*`'`	Load up to eight ASCII characters into C.	`LCHEX 'AB'`

10.2.3 P Register Instructions

In the following instructions,

- *n* is an expression whose hex value is from 0 through F

The C register is the only working register used with the P register. All arithmetic calculations on the pointer are performed in HEX mode. During calculation operations, the carry flag will be set if the calculation overflows or borrows, otherwise the carry flag will be cleared.

Instruction	Description	Examples
`P=` *n*	Set P register to *n*	`P= 6`
`P=P+1`	Increment P register	`P=P+1`
`P=P-1`	Decrement P register	`P=P-1`
`C+P+1`	Add P register plus one to A field in C	`C+P+1`
`CPEX` *n*	Exchange P register with nibble *n* in C	`CPEX 15`
`P=C` *n*	Copy nibble *n* in C to P register	`P=C 2`
`C=P` *n*	Copy P register to nibble *n* in C	`C=P 0`

10.2.4 Scratch Register Instructions

In the following instructions,

- *r* = A or C
- *ss* = R0, R1, R2, R3, or R4
- *fs* = A, P, WP, XS, X, S, M, B, W, or a number *n* from 1 through 16

Instruction	Description	Examples
r`=`*ss*	Copy *ss* into *r*	`C=R4`
ss`=`*r*	Copy *r* into *ss*	`R0=A`
rss`EX`	Exchange *r* and *ss*	`AR1EX`
r`=`*ss*`.F` *fs*	Copy *ss*(*fs*) to *r*(*fs*)	`A=R0.F A`
ss`=`*r*`.F` *fs*	Copy *r*(*fs*) to *ss*(*fs*)	`R3=C.F M`
rss`EX.F` *fs*	Exchange *r*(*fs*) with *ss*(*fs*)	`CR2EX.F B`

10.2.5 Shift Instructions

In the following instructions,

- *r* = A, B, C, or D

- *fs* = A, P, WP, XS, X, S, M, B, or W

Non-circular shift operations shift in zeros. If any shift-right operation, circular or non-circular, moves a non-zero nibble or bit from the right end of a register or field, the Sticky Bit SB is set. The Sticky Bit is cleared only by a `SB=0` or `CLRHST` instruction.

Instruction		Description	Examples
*r*SRB		Shift *r* right by one bit	`ASRB`
*r*SRB.F	*fs*	Shift *fs* field in *r* right by one bit	`CSRB.F A`
*r*SLC		Shift *r* left by one nibble (circular)	`BSLC`
*r*SRC		Shift *r* right by one nibble (circular)	`CSRC`
*r*SL	*fs*	Shift *fs* field in *r* left by one nibble	`DSL M`
*r*SR	*fs*	Shift *fs* field in *r* right by one nibble	`ASR A`

10.2.6 Logical Instructions

In the following instructions,

- (*r*, *s*) = (A, B), (A, C), (B, A), (B, C), (C, A), (C, B), (C, D), or (D, C)
- *fs* = A, P, WP, XS, X, S, M, B, or W

Instruction		Description	Examples
r=*r*&*s*	*fs*	*fs* field in *r* AND *fs* field in *s* into *fs* field in *r*	`A=A&C A`
r=*r*!*s*	*fs*	*fs* field in *r* OR *fs* field in *s* into *fs* field in *r*	`D=D!C XS`

Note that XOR is missing. The following four instructions implement A XOR C in the A field:

```
B=A   A  Save a copy of A
B=B&C A  A AND C
A=A!C A  A OR C
A=A-B A  A XOR C = (A OR C) - (A AND C)
```

10.2.7 Arithmetic Instructions

Arithmetic results depend on the current arithmetic mode. In HEX mode (set by `SETHEX`), nibble values range from 0 through F. In decimal mode (set by `SETDEC`), nibble values range from 0 through 9, and arithmetic is BCD arithmetic.

There are two groups of arithmetic instructions. In the first group (general), almost all combinations of the four working registers are possible; in the second group (restricted), only a few combinations are possible. During those operations that involve a calculation, the carry flag is set if the calculation overflows or borrows; otherwise the carry flag is cleared.

General Arithmetic Instructions.

In the following instructions,

- (*r*, *s*) = (A, B), (A, C), (B, A), (B, C), (C, A), (C, B), (C, D), or (D, C)
- *fs* = A, P, WP, XS, X, S, M, B, or W

Instruction	Description	Examples
r=0 *fs*	Set *fs* field in *r* to zero	C=0 W
r=*s* *fs*	Copy *fs* field in s into *fs* field in *r*	A=C A
s=*r* *fs*	Copy *fs* field in *r* into *fs* field in *s*	C=A A
*rs*EX *fs*	Exchange *fs* field in *r* and *fs* field in *s*	ACEX A
r=*r*+*r* *fs*	Double *fs* field in *r* (shift left by one bit)	A=A+A A
r=*r*+1 *fs*	Increment *fs* field in *r* by 1	C=C+1 B
r=*r*-1 *fs*	Decrement *fs* field in *r* by 1	C=C-1 B
r=*r*+CON *fs*,*d*	Add constant *d* to field *fs* in *r*	A=A+CON A,5
r=*r*-CON *fs*,*d*	Subtract constant *d* from field *fs* in *r*	C=C-CON A,10
r=-*r* *fs*	Tens complement or twos complement, depending on arithmetic mode, of *fs* field in *r*. Clears carry if *r*(*fs*) was zero, otherwise sets carry.	C=-C S
r=-*r*-1 *fs*	Nines complement or ones complement, depending on arithmetic mode, of *fs* field in *r*. Clears carry unconditionally.	C=-C-1 S
r=*r*+*s* *fs*	Sum fs field in *r* and *fs* field in *s* into *fs* field in *r*	C=C+A A
s=*r*+*s* *fs*	Sum fs field in *r* and *fs* field in *s* into *fs* field in *s*	A=C+A A

Restricted Arithmetic Instructions.

In the following instructions,

- (*r*, *s*) = (A, B), (B, C), (C, A), or (D, C)
- *fs* = A, P, WP, XS, X, S, M, B, or W

Instruction	Description	Examples
r=*r*-*s* *fs*	Difference of *fs* field in *r* and *fs* field in *s* into *fs* field in *r*	A=A-B A
r=*s*-*r* *fs*	Difference of *fs* field in *s* and *fs* field in *r* into *fs* field in *r*	B=C-B A
s=*s*-*r* *fs*	Difference of *fs* field in *s* and *fs* field in *r* into *fs* field in *s*	A=A-C A

10.2.8 Branching Instructions

GOTO and GOSUB Instructions.

In the following instructions,

- *label* is a symbol defined in the label field of an instruction within the current code object
- =*label* is an entry in the lower 256K of the HP 48 operating system
- *offset* is the distance in nibbles to the specified *label*
- *r* = A or C

Instruction	Description	Examples
`GOTO` *label*	Short relative jump (–2047 ≤ offset ≤ 2048)	`GOTO LBL01`
`GOYES` *label*	Short relative jump if test is true (–125 ≤ offset ≤ 130)	`?A=C A` `GOYES DoEqual`
`GOC` *label*	Short relative jump if carry set (–127 ≤ offset ≤ 128)	`GOC Done`
`GONC` *label*	Short relative jump if carry clear (–127 ≤ offset ≤ 128)	`GONC NotDone`
`GOLONG` *label*	Long relative jump (–32762 ≤ offset ≤ 32768)	`GOLONG End`
`GOVLNG` =*label*	Absolute jump	`GOVLNG =PUSH#ALOOP`
`GOSUB` *label*	Short relative subroutine jump (–2044 ≤ offset ≤ 2051)	`GOSUB parse`
`GOSUBL` *label*	Long relative subroutine jump (–32762 ≤ offset ≤ 32773)	`GOSUBL output`
`GOSBVL` =*label*	Absolute subroutine jump	`GOSBVL =POP#A`
`PC=`*r*	Direct jump to address in *r*[A]	`PC=A`
r`=PC`	Copies the PC to *r*[A]	`C=PC`
r`PCEX`	Direct jump to *r*[A], saving PC in *r*[A]	`APCEX`
`PC=(`*r*`)`	Indirect jump: *r*[A] points to the address to jump to	`PC=(C)`

Note: All calls to HP 48 entries from code objects should use `GOVLNG` or `GOSBVL`.

Return Instructions

Instruction	Description	Examples
`RTN`	Return	`RTN`
`RTNSC`	Return and set carry	`RTNSC`
`RTNCC`	Return and clear carry	`RTNCC`
`RTNSXM`	Return and set XM status bit	`RTNSXM`
`RTI`	Return from interrupt (enable interrupts)	`RTI`
`RTNC`	Return if carry set	`RTNC`
`RTNNC`	Return if no carry set	`RTNNC`
`RTNYES`	Return if test is true (used only with test instructions)	`?ST=0 1` `RTNYES`

Return Stack Instructions

Instruction	Description	Examples
`RSTK=C`	Push A field in C onto return stack	`RSTK=C`
`C=RSTK`	Pop return stack into A field in C	`C=RSTK`

10.2.9 Test Instructions

Each test instruction must be followed by a `GOYES` or a `RTNYES` instruction. The test instruction and the `GOYES` or `RTNYES` instruction combine to generate a single opcode. Each test will set the carry flag if true, or clear the carry flag if false. All tests are unsigned and performed only on the selected field.

Register Tests.

In the following instructions,

- `(`*r*`, `*s*`)` = (A, B), (A, C), (B, A), (B, C), (C, A), (C, B), (C, D), or (D, C)
- *fs* = A, P, WP, XS, X, S, M, B, or W

Instruction	Description	Examples
?*r*=*s* *fs*	Is *fs* field in requal to *fs* field of *s*?	`?B=C A` `GOYES ItIs`
?*r*#*s* *fs*	Is *fs* field in *r* not equal to *fs* field of *s*?	`?C#D S` `GOYES CDSNotEqual`
?*r*=0 *fs*	Is *fs* field in *r* equal to zero?	`?B=0 P` `RTNYES`
?*r*#0 *fs*	Is *fs* field in *r* not equal to zero?	`?B#0 P` `RTNYES`
?*r*>*s* *fs*	Is *fs* field in *r* greater than *fs* field of *s*?	`?A>C A` `GOYES Bigger`
?*r*<*s* *fs*	Is *fs* field in *r* less than *fs* field of *s*?	`?A<C A` `GOYES Smaller`
?*r*=*s* *fs*	Is *fs* field in *r* greater than or equal to *fs* field of *s*?	`?B>=C WP` `GOYES GThanE`
?*r*<=*s* *fs*	Is *fs* field in *r* less than or equal to *fs* field of *s*?	`?B<=C WP` `GOYES LThanE`

Register Bit Tests.

In the following instructions,

- *n* is an expression whose hex value is from 0 through F
- *r* = A or C

Instruction	Description	Examples
?*r*BIT=0 *n*	Is bit *n* in *r* equal to 0?	`?ABIT=0 2` `RTNYES`
?*r*BIT=1 *n*	Is bit *n* in *r* equal to 1?	`?CBIT=1 15` `RTNYES`

Pointer Tests.

In the following instructions,

- *n* is an expression whose hex value is from 0 through F

Instruction	Description	Examples
?P= *n*	Is P register equal to *n*?	`?P= 0` `GOYES Done`
?P# *n*	Is P register not equal to *n*?	`?P# 0` `GOYES NotDone`

Program Status Bit Tests.

In the following instructions,

- *n* is an expression whose hex value is from 0 through F

Instruction	Description	Examples
`?ST=0` n	Is bit n in ST equal to 0?	`?ST=0 0` `RTNYES`
`?ST=1` n	Is bit n in ST equal to 1?	`?ST=1 1` `GOYES TryAgain`
`?ST#0` n	Is bit n in ST not equal to 0?	`?ST#0 6` `GOYES TryOver`
`?ST#1` n	Is bit n in ST not equal to 1?	`?ST#1 3` `RTNYES`

Hardware Status Bit Tests.

Instruction	Description	Examples
`?XM=0`	Is the External Module Missing bit clear?	`?XM=0` `RTNYES`
`?SB=0`	Is the Sticky Bit clear?	`?SB=0` `GOYES NotShifted`
`?SR=0`	Is the Service Request bit clear?	`?SR=0` `RTNYES`
`?MP=0`	Is the Module Pulled bit clear?	`?MP=0` `GOYES MPClear`

10.2.10 Register & Status Bit Instructions

Register Bit Instructions.

In the following instructions,

- n is an expression whose hex value is from 0 through F
- r = A or C

Instruction	Description	Examples
r`BIT=0` n	Clear bit n in r	`ABIT=0 0`
r`BIT=1` n	Set bit n in r	`CBIT=1 9`

Program Status Bit Instructions.

In the following instructions,

- n is an expression whose hex value is from 0 through F

Instruction	Description	Examples
`ST=0` n	Clear bit n in ST	`ST=0 0`
`ST=1` n	Set bit n in ST	`ST=1 4`
`CSTEX`	Exchange X field in C and bits 0 through 11 in ST	`CSTEX`
`C=ST`	Copy bits 0 through 11 in ST into X field in C	`C=ST`
`ST=C`	Copy X field in C into bits 0 through 11 in ST	`ST=C`
`CLRST`	Clear bits 0 through 11 in ST	`CLRST`

Hardware Status Bit Instructions.

Instruction	Description	Examples
`SB=0`	Clear Sticky Bit (SB)	`SB=0`
`SR=0`	Clear Service Request (SR) bit	`SR=0`
`MP=0`	Clear Module Pulled (MP) bit	`MP=0`
`XM=0`	Clear External Module (XM) bit	`XM=0`
`CLRHST`	Clear SB, SR, MP, and XM bits	`CLRHST`

10.2.11 System Control Instructions

Instruction	Description	Examples
`SETHEX`	Set arithmetic mode to hexadecimal	`SETHEX`
`SETDEC`	Set arithmetic mode to decimal	`SETDEC`
`CONFIG`	Configure a device to the address in C(A)	`CONFIG`
`UNCNFG`	Unconfigure a device at address in C(A)	`UNCNFG`
`RESET`	Send Reset command to the system bus	`RESET`
`BUSCB`	Issue bus command B	`BUSCB`
`BUSCC`	Issue bus command C	`BUSCC`
`BUSCD`	Issue bus command D	`BUSCD`
`SHUTDN`	Stop CPU, stay in low-power mode until wake-up	`SHUTDN`
`C=ID`	Copy chip ID from system bus to C(A)	`C=ID`
`SREQ?`	Set C(0) to service request response from bus, set SR if service requested	`SREQ?`
`INTOFF`	Disable maskable interrupts	`INTOFF`
`INTON`	Enable maskable interrupts	`INTON`

10.2.12 Keyscan Instructions

Instruction	Description	Examples
`OUT=C`	Copy X field in C into OUT	`OUT=C`
`OUT=CS`	Copy nibble 0 of C into OUT	`OUT=CS`
`A=IN`	Copy IN into nibbles 0 through 3 in A	`A=IN`
`C=IN`	Copy IN into nibbles 0 through 3 in C	`C=IN`

Note that `A=IN` and `C=IN` must be executed on an even address. An reliable way to do this is to call the entries `AINRTN` and `CINRTN`, illustrated in *Keyboard Scanning* on page 232.

10.2.13 NOP Instructions

Instruction	Description	Examples
`NOP3`	Three-nibble no-op	`NOP3`
`NOP4`	Four-nibble no-op	`NOP4`
`NOP5`	Five-nibble no-op	`NOP5`

10.2.14 Assembler Pseudo-Op Instructions

The following pseudo-ops are a few of the pseudo-ops available in the SASM assembler.

Data Storage and Allocation.

In the following instructions,

- *nnnnn* is an expression whose hex value is from 0 through FFFFF
- *expr* is an expression that evaluates to a constant from 0 through FFFFF
- *m* is a one digit decimal integer constant
- *label* is a symbol defined in the label field of an instruction within the current code object
- *h* is a hex digit

Instruction	Description	Examples
`BSS` *nnnnn*	Allocate *nnnnn* zero nibbles here. *Note: Do not write self-modifying code objects that will be used in a library in the HP 48! (The library checksums will become invalid.)*	`BSS 4`
`CON(`*m*`) expr`	Generate an *m* nibble constant	`CON(5) =DOCOL`
`REL(`*m*`) label`	Generate an *m* nibble relative offset	`REL(5) =EndGrob`
`NIBASC \ascii\`	Generate up to 40 ASCII characters. Each character has the nibbles reversed.	`NIBASC \Fred\`
`NIBHEX` *h ... h*	Generate up to 80 hex digits	`NIBHEX 1424FC`

Symbol Definition.

In the following instructions,

- *symbol* is a name for an address, defined in the label field of an instruction (global if preceded with =)
- *expr* is an expression that evaluates to a constant from 0 through FFFFF

Instruction	Description	Examples
symbol `EQU` *expr*	Assigns the value *expr* to symbol. If *symbol* is already defined, an error is generated.	`size EQU 232` `=SEMI EQU #0312B`
symbol `=` *expr*	Assigns the value *expr* to *symbol*. Replaces any existing value.	`size = 233`

Chapter 11

Writing Your Own Code Objects

Assembly language code is encapsulated in a *code object* (type 25), which is one of the object types that the HP 48 recognizes. In this chapter we'll introduce a few ways to write your own code objects.

11.1 Code Object Execution

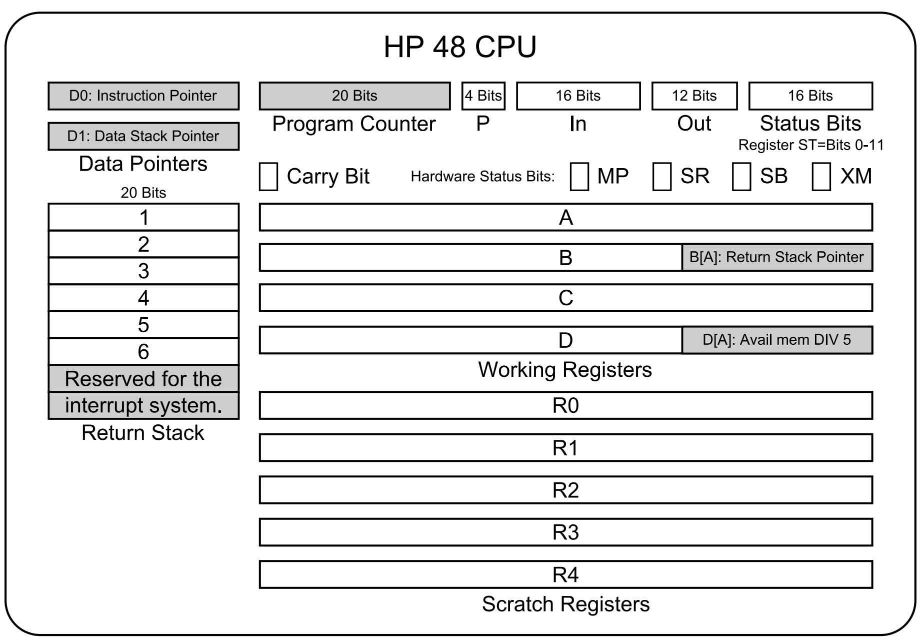
When a code object begins to execute, it must account for information vital to System-RPL execution that resides in the CPU. Four registers in the CPU contain this information, usually known as the "RPL pointers":

D0 The instruction pointer

D1 The data stack pointer

B[A] The return stack pointer

D[A] (Available memory) DIV 5



In addition to the information in the registers described above, P is guaranteed to be 0 and the CPU is in HEX mode. Both of these conditions *must* also be true when the code object terminates and the system returns to RPL execution. There are two common ways to terminate code object execution and resume execution of the RPL inner loop:

- Resume execution at the pointee of the top of the return stack:

`A=DAT0`	`A`	*Read the pointer to the next RPL object to be executed*
`D0=D0+`	`5`	*Advance the instruction pointer*
`PC=(A)`		*Branch to the next instruction*

  The example programs SWP and DRP9 illustrate this technique.

- Resume execution via another object. This example returns to RPL via TRUE:

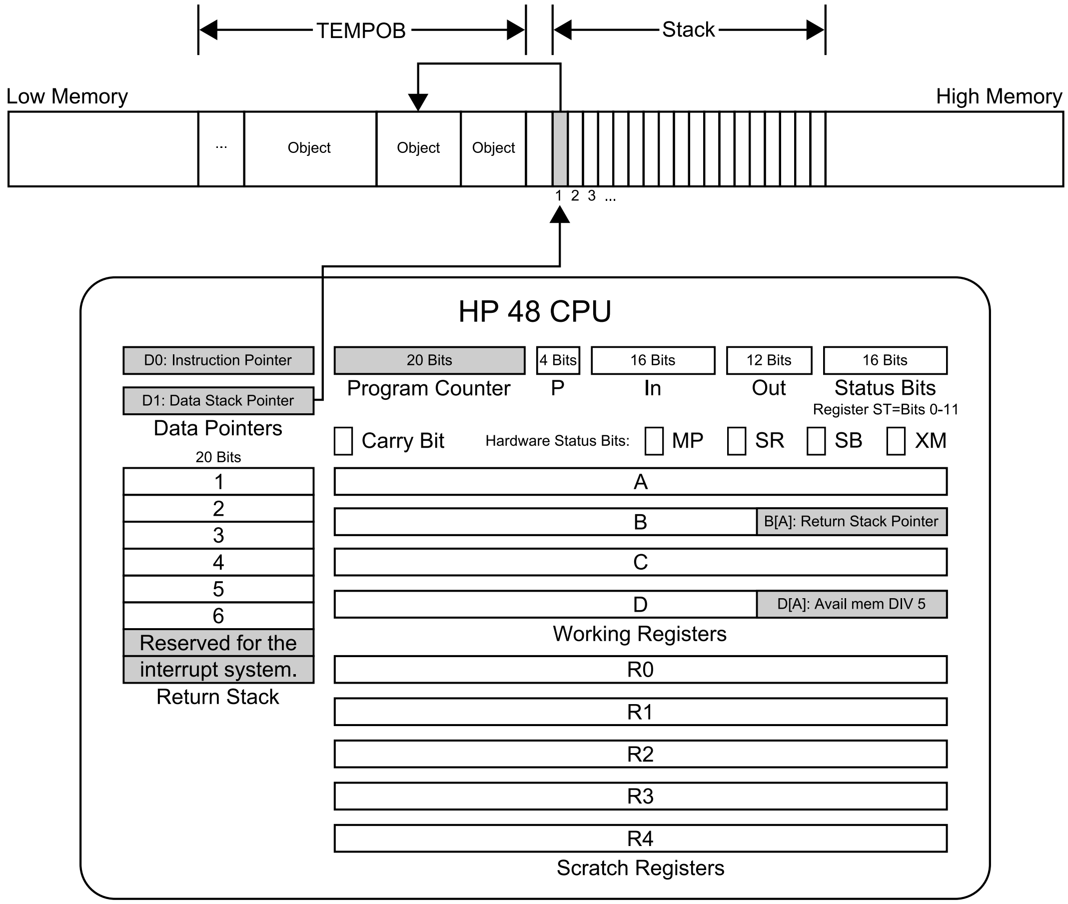
`LC(5)`	`=TRUE`	*Load the address of the object to execute*
`A=C`	`A`	*Copy to A*
`PC=(A)`		*Branch to TRUE*

  The example program ABSF illustrates this technique.

Many code objects will take their arguments from the stack (via D1), save the RPL pointers, perform their task, then restore the RPL pointers before returning to RPL execution. The entries `SAVPTR` and `GETPTR` may be used to save the contents of D0, D1, B[A], and D[A] in reserved RAM locations and restore them later, thus freeing the entire CPU for use by an application.

11.2 Stack Access

Stack manipulation tasks provide one way to introduce some simple tasks that do not require `SAVPTR` and `GETPTR`, so we begin by illustrating some simple stack operations. We begin by illustrating the pointer path from CPU register D1 to the actual object in memory:



The contents of D1 point to a series of 5-nibble stack pointers, each of which in turn point to the actual objects. Note that TEMPOB is not the only place a stack pointer can point to — user variable memory is another possible destination, and the differences are important. Stack pointers can also point to objects like the display grobs and temporary environments.

11.2.1 Example: SWAP Two Objects

The program SWP is the first example — it swaps the top two objects on the stack in exactly the same manner as the built-in SWAP command. Notice that A and C are used (so B and D are not disturbed), and that D1 is restored to its original value. Notice that only the pointers are shifted — the objects themselves do not move.

```
SWP 26.5 Bytes Checksum #D1C0h
( ob1 ob2 → ob2 ob1 )
          NIBASC  /HPHP48-A/  This is a download header for binary transfer to the HP 48
          CON(5)  =DOCODE     This is the prologue for a code object
          REL(5)  end         The length field — indicates the size of the code object
          A=DAT1  A           Copy the stack level 1 pointer to A[A]
          D1=D1+  5           Advance D1 to stack level 2
          C=DAT1  A           Copy the stack level 2 pointer to C[A]
          DAT1=A  A           Replace stack level 2 with the original stack level 1 pointer
          D1=D1-  5           Move D1 back to stack level 1
          DAT1=C  A           Replace stack level 1 with the original stack level 2 pointer
                              The next three instructions embody the RPL inner loop:
          A=DAT0  A           Read the pointer to the next RPL object to be executed
          D0=D0+  5           Advance the instruction pointer
          PC=(A)              Branch to the next instruction
end
```

11.2.2 Example: DROP Nine Objects

The program DRP9 drops nine objects from the stack very quickly. Dropping an object is very simple — simply increment the top-of-stack pointer D1 by five nibbles and update the available memory stored in D[A]. Assuming there are no other stack pointers to the discarded object and the discarded object is in temporary memory (TEMPOB), the object is effectively "orphaned" and its memory will be recovered during the next garbage collection.

DRP9 also illustrates the use of a counter and the `GONC` instruction. We use the P register for the counter in this example for several reasons:

- P is optimal for counting applications where no more than 16 repetitions are required. (Be sure that a non-zero value of P during the loop won't adversely affect data loading instructions like `LCHEX`.)
- Incrementing P is fast — taking only 3 cycles.
- When P is used for the counter, it is not neccessary to consume part of a working register for the counter.

This example could also be coded using P as a count*down* counter, but the value of P would be 15 at the end, then a P=0 instruction would have to be added for a safe exit back to RPL.

```
DRP9 24.5 Bytes Checksum #8093h
( ob1 ...  ob9 → )
          NIBASC  /HPHP48-A/  This is a download header for binary transfer to the HP 48
          CON(5)  =DOCODE     This is the prologue for a code object
          REL(5)  end         The length field — indicates the size of the code object
          P=      16-9        P will be used as a counter — we'll count "up to 0"
LoopTop                       This label marks the top of the drop loop
          D1=D1+  5           Advance D1 to the next stack level
          D=D+1   A           Increment available memory
          P=P+1               Increment the counter
          GONC    LoopTop     If no carry, there's more stack levels to do so branch to LoopTop
                              If carry is set, we're done and P=0 (wrapped from F)
                              The next three instructions embody the RPL inner loop:
          A=DAT0  A           Read the pointer to the next RPL object to be executed
          D0=D0+  5           Advance the instruction pointer
          PC=(A)              Branch to the next instruction
end
```

11.3 Reading Assembly Language Entry Descriptions

The entries described here require specific conditions to be met in order to be used successfully. The entry and exit conditions refer to the following criteria:

- The location of the RPL pointers — either in the CPU or saved in RAM.
- The arithmetic mode — HEX or DEC.
- Contents of various registers
- The state of the carry flag — CS = carry set, CC = carry clear
- The number of stack levels used by the routine (you should never use more than 6)

Unless stated otherwise, it is always assumed that the CPU is in HEX mode and register P is 0.

Most entries are called with `GOSBVL`, but some entries (like `GETPTRLOOP`) never return, since they restart the RPL inner loop. The "Call with" entry in these descriptions suggests which type of call to use.

11.4 Saving and Restoring the RPL Pointers

The RPL pointers can be saved in reserved RAM locations by calling `SAVPTR` and restored by calling `GETPTR`.

`SAVPTR`	#0679Bh
Saves D0, D1, B[A], and D[A] in reserved memory	
Entry:	RPL pointers in the CPU
Call with:	`GOSBVL`
Exit:	RPL pointers saved. D1, A[A], B[A], and D[A] are unchanged
Uses:	D0, D1, B[A], C[A], D[A]
Stack Levels:	0

`GETPTR`	#067D2h
Restores D0, D1, B[A], and D[A] from reserved memory	
Entry:	RPL pointers saved
Call with:	`GOSBVL`
Exit:	RPL pointers in CPU.
Uses:	D0, D1, B[A], C[A], D[A]
Stack Levels:	0

There are several entry points which combine the process of restoring the RPL pointers and returning to RPL execution, sometimes returning objects to the stack in the process. The most basic of these entries is `GETPTRLOOP`, which has the following entry and exit conditions:

`GETPTRLOOP`	#05143h
Restores D0, D1, B[A], and D[A] from reserved memory, then restarts the RPL inner loop	
Entry:	RPL pointers saved
Call with:	`GOVLNG`
Exit:	To RPL
Uses:	D0, D1, B[A], C[A], D[A]
Stack Levels:	0

11.4.1 Example: Reversing Objects on the Stack

The program RVRSO reverses N objects on the stack, where N is a real number indicating the number of objects to reverse. The source code illustrates a typical mix of System-RPL and assembler code to accomplish a task. The System-RPL shell validates the input arguments, while the assembly language code does the actual work of reversing a series of stack pointers.

RVRSO 75.5 Bytes Checksum #8501h

(ob_1 ... ob_n N → ob_n ... ob_1 N)

```
ASSEMBLE
        NIBASC /HPHP48-A/       This is a download header for binary transfer to the
                                HP 48
RPL
::
  OLASTOWDOB! CKNNOLASTWD       Validate the number of arguments on the stack
  ONE OVER #< IT                If there's at least two objects on the stack, execute the
                                code object
CODE
        GOSBVL  =SAVPTR         Save the RPL pointers in RAM
        GOSBVL  =POP#           A[A] = number of objects on the stack
        C=A     A               #items in C[A]
        C=C+C   A               #items * 2
        C=C+C   A               #items * 4
        C=C+A   A               C[A] = #items*5
        B=0     W               Zero out entire B register
        B=A     A               B[A] = count
        BSRB    A               Divide #items by 2
        AD1EX                   A → first item on stack
        D1=A                    D1 → first item on stack
        A=A+C   A               A[A] → past last item
        D0=A                    D0 → past last item
        D0=D0-  5               D0 → last item
RvrTop
        B=B-1   A               Decrement counter
        GOC     RvrBot          If carries, no more pairs to reverse
        A=DAT0  A               Read first item
        C=DAT1  A               Read last item
        DAT0=C  A               Write last item in first item's original location
        DAT1=A  A               Write first item in last item's original location
        D1=D1+  5               Move D1 to next pointer location
        D0=D0-  5               Move D0 to previous pointer location
        GONC    RvrTop          (BET) Branch every time to RvrTop
*
RvrBot
        GOVLNG  =GETPTRLOOP     Restore pointers, return to RPL
ENDCODE
  UNCOERCE                      Convert #objects back into real number
;
```

There are two notation habits used in this listing to help understand the code. The first is the use of "(BET)" in the branch to `RvrTop`. (BET) stands for "Branch Every Time" an unconditional branch. This tells a reader that you intend this to be an unconditional branch, and is usually used where a branch is dependent on the state of the carry flag. There is no need to use (BET) for a `GOTO` instruction. The other notation is the placement of an asterisk (*) above the label `RvrBot`. This is used to indicate that control flow to the following label must be from a jump instruction, and *cannot* flow from previous instructions.

11.4.2 Example: Clearing A Grob

This example might also live in a graphics discussion, but it's a good way to get some practice with counters and a simple way to save just one of the RPL pointers. The following code object uses D1, A[W], C[A], and one level of the return stack to clear a grob.

To understand this code object, note the structure of a grob object:

Prologue	Length	Height	Width	Body

The prologue, length, height, and width fields are 5 nibbles each. The length field contains a self-relative length to the end of the body. This means the length field is always at least 15, to account for the size of the length, height, and width fields.

Notice that this object drops the grob pointer from the stack. If you don't want the pointer dropped, just leave out the two instructions that increment D1 and update D[A].

```
CLGRB 56.5 Bytes Checksum #E4D0h
( grob → )
NIBASC      /HPHP48-A/
CON(5)      =DOCODE
REL(5)      end
            A=DAT1   A          A → grob
*
* Optional:  The next two instructions pop the grob pointer
*
            D1=D1+   5          Pop grob: first advance stack pointer
            D=D+1    A          then increment available mem DIV 5
*
            CD1EX               C[A] = updated stack pointer
            D1=A                D1 → grob prologue
            RSTK=C              Save D1 on return stack
            D1=D1+   5          D1 → grob length
            A=DAT1   A          A[A] = grob length
            LC(5)    15         Length of length field, height, width
            C=A-C    A          C[A] = number of nibbles to clear
            D1=D1+   15         Point D1 to first nibble of grob body
            C=C-1    A          Decrement length to option base 0
            GOC      quit       If zero length, quit
            A=0      W          Clear A to write zeros
            P=C      0          P = (length MOD 16)-1
            CSR      A          Divide length by 16 to create block counter
nxtblk
            C=C-1    A          Decrement block counter
            GOC      rest       If carries here, no more blocks to write
            DAT1=A   W          Write a block of 16 zeros
            D1=D1+   16         Advance write pointer
            GONC     nxtblk     (BET) Go see if there's more blocks to do
*
rest
            DAT1=A   WP         Write partial block
            P=       0          Reset P
quit
            C=RSTK              Recover stack pointer
            D1=C                and put it back into D1
            A=DAT0   A          Read pointer to next object in runstream
            D0=D0+   5          Advance instruction pointer
            PC=(A)              Branch to next instruction
end
```

11.5 Stack Utilities

The entries described here are useful for popping objects from the stack or pushing objects on the stack.

11.5.1 Pop Utilities

While you can follow the stack pointer to the object directly in memory, remember that small bint values and some real numbers can be represented by pointers to objects in ROM. It's safer to pop the values into the CPU.

`POP#`	#06641h
Pops a bint from the stack	
Entry:	(# →) RPL pointers in the CPU
Call with:	`GOSBVL`
Exit:	A[A]=#, updated RPL pointers in the CPU
Uses:	C[A]
Stack Levels:	0

`POP2#`	#03F5Dh
Pops two bints from the stack	
Entry:	($\#_2$ $\#_1$ →) RPL pointers in the CPU
Call with:	`GOSBVL`
Exit:	A[A]=$\#_2$, C[A]=$\#_1$, updated RPL pointers in the CPU
Uses:	C[A]
Stack Levels:	1

`POP1%`	#29FDAh
Pops a real number from the stack	
Entry:	(% →) RPL pointers in the CPU
Call with:	`GOSBVL`
Exit:	A[W]=%, RPL pointers saved, **DEC mode**
Uses:	C[A], D[A], D0, D1
Stack Levels:	0

`POP2%`	#2A002h
Pops two real numbers from the stack	
Entry:	($\%_2$ $\%_1$ →) RPL pointers in the CPU
Call with:	`GOSBVL`
Exit:	A[W]=$\#_2$, C[W]=$\#_1$, RPL pointers saved, **DEC mode**
Uses:	D[A], D0, D1
Stack Levels:	0

`popflag`	#61A02h
Pops a flag from the stack, sets carry if flag was TRUE	
Entry:	(FLAG →) RPL pointers in the CPU
Call with:	`GOSBVL`
Exit:	CS if flag=TRUE, RPL pointers in the CPU
Uses:	A[A], C [A]
Stack Levels:	0

`PopASavptr`	#3251Ch
Pops an object from the stack, saves pointers	
Entry:	(ob →) RPL pointers in the CPU
Call with:	`GOSBVL`
Exit:	A[A]→ob, RPL pointers saved
Uses:	A[A], C[A]
Stack Levels:	0

11.5.2 Push Utilities

The push utilities execute fairly quickly and use few registers *unless* a garbage collection is needed. The register usage and stack level usage below reflects the worst-case scenario — a trip through garbage collection. There are a wide variety of flag utilities — there should be one to suit every need.

Bints

`PUSHA`	#03A86h
Pushes a pointer to an object on the stack and restarts the RPL inner loop. *Note:* The pointer *must not* reference an object in TEMPOB.	
Entry:	A[A] → object, RPL pointers in the CPU
Call with:	`GOVLNG`
Exit:	(→ ob) To RPL

`PUSH#`	#06537h
Pushes a bint on the stack	
Entry:	R0[A]=#, RPL pointers saved
Call with:	`GOSBVL`
Exit:	(→ #), updated RPL pointers in the CPU
Uses:	A[W], B[W], C[W], D[W], ST[0], ST[10]
Stack Levels:	3

`PUSH#LOOP`	#0357Fh
Pushes a bint on the stack, restarts the RPL inner loop	
Entry:	R0[A]=#, RPL pointers saved
Call with:	`GOVLNG`
Exit:	(→ #) To RPL

`PUSH#ALOOP`	#0357Ch
Pushes a bint on the stack, restarts the RPL inner loop	
Entry:	A[A]=#, RPL pointers saved
Call with:	`GOVLNG`
Exit:	(→ #) To RPL

`PUSH2#`	#06529h
Pushes two bints on the stack	
Entry:	R0 [A] =$\#_1$, R1 [A] =$\#_2$ RPL pointers saved
Call with:	`GOSBVL`
Exit:	(→ $\#_1$ $\#_2$), updated RPL pointers in the CPU
Uses:	A[W], B[W], C[W], D[W], ST[0], ST[10]
Stack Levels:	4

Real Numbers

`PUSH%`	#2A188h
Sets HEX mode, pushes a real number on the stack	
Entry:	A[W]=%, RPL pointers saved
Call with:	`GOSBVL`
Exit:	(→ %), updated RPL pointers in the CPU
Uses:	A[W], B[W], C[W], D[W], ST[0], ST[10]
Stack Levels:	3

`PUSH%LOOP`	#2A23Dh
Sets HEX mode, pushes a real number on the stack, restarts the RPL inner loop	
Entry:	A[W]=%, RPL pointers saved
Call with:	`GOSBVL`
Exit:	(→ %), To RPL
Uses:	A[W], B[W], C[W], D[W], ST[0], ST[10]
Stack Levels:	3

Flags

`GPOverWrTLp`	#62076h
Restores the RPL pointers, overwrites stack level 1 with TRUE, restarts the RPL inner loop	
Entry:	(ob →) RPL pointers saved
Call with:	`GOVLNG`
Exit:	(→ TRUE), To RPL

`GPOverWrFLp`	#62096h
Restores the RPL pointers, overwrites stack level 1 with FALSE, restarts the RPL inner loop	
Entry:	(ob →) RPL pointers saved
Call with:	`GOVLNG`
Exit:	(→ FALSE), To RPL

`GPOverWrT/FL`	#62073h
Restores the RPL pointers, overwrites stack level 1 with carry-specified flag, restarts the RPL inner loop	
Entry:	(ob →) RPL pointers saved, Carry: set=TRUE, clear=FALSE
Call with:	`GOVLNG`
Exit:	(→ FLAG), To RPL

`GPPushTLoop`	#620B9h
Restores the RPL pointers, pushes TRUE on the stack, restarts the RPL inner loop	
Entry:	RPL pointers saved
Call with:	`GOVLNG`
Exit:	(→ TRUE), To RPL

`GPPushFLoop`	#620D2h
Restores the RPL pointers, pushes FALSE on the stack, restarts the RPL inner loop	
Entry:	RPL pointers saved
Call with:	`GOVLNG`
Exit:	(→ FALSE), To RPL

`GPPushT/FLp`	#620B6h
Restores the RPL pointers, pushes carry-specified flag on the stack, restarts the RPL inner loop	
Entry:	RPL pointers saved, Carry: set=TRUE, clear=FALSE
Call with:	`GOVLNG`
Exit:	(→ FLAG), To RPL

`OverWrTLoop`	#62080h
Overwrites stack level 1 with TRUE, restarts the RPL inner loop	
Entry:	(ob →) RPL pointers in CPU
Call with:	`GOVLNG`
Exit:	(→ TRUE), To RPL

`OverWrFLoop`	#620A0h
Overwrites stack level 1 with FALSE, restarts the RPL inner loop	
Entry:	(ob →) RPL pointers in CPU
Call with:	`GOVLNG`
Exit:	(→ FALSE), To RPL

`OverWrT/FLp`	#6209Dh
Overwrites stack level 1 with carry-specified flag, restarts the RPL inner loop	
Entry:	(ob →) RPL pointers in CPU, Carry: set=TRUE, clear=FALSE
Call with:	`GOVLNG`
Exit:	(→ FLAG), To RPL

`OverWrF/TLp`	#6207Dh
Overwrites stack level 1 with carry-specified flag, restarts the RPL inner loop	
Entry:	(ob →) RPL pointers in CPU, Carry: set=FALSE, clear=TRUE
Call with:	`GOVLNG`
Exit:	(→ FLAG), To RPL

`PushTLoop`	#620C3h
Pushes TRUE, restarts the RPL inner loop	
Entry:	RPL pointers in CPU
Call with:	`GOVLNG`
Exit:	(→ TRUE), To RPL

`PushFLoop`	#620DCh
Pushes FALSE, restarts the RPL inner loop	
Entry:	RPL pointers in CPU
Call with:	`GOVLNG`
Exit:	(→ FALSE), To RPL

PushT/FLoop	#620D9h
Pushes carry-specified flag, restarts the RPL inner loop	
Entry:	RPL pointers in CPU, Carry: set=TRUE, clear=FALSE
Call with:	GOVLNG
Exit:	(→ FLAG), To RPL

PushF/TLoop	#62000h
Overwrites stack level 1 with carry-specified flag, restarts the RPL inner loop	
Entry:	RPL pointers in CPU, Carry: set=FALSE, clear=TRUE
Call with:	GOVLNG
Exit:	(→ FLAG), To RPL

Arbitrary Objects

GPOverWrR0Lp	#0366Fh
Restores the RPL pointers, overwrites stack level 1 with R0[A], restarts the RPL inner loop	
Entry:	(ob_{any} →) RPL pointers saved
Call with:	GOVLNG
Exit:	(→ $ob_{R0[A]}$), To RPL

GPOverWrALp	#03672h
Restores the RPL pointers, overwrites stack level 1 with A[A], restarts the RPL inner loop	
Entry:	(ob_{any} →) RPL pointers saved
Call with:	GOVLNG
Exit :	(→ $ob_{A[A]}$) , To RPL

11.5.3 Examples: Indicated ABS

The code object ABSF pops a real number from the stack and tests the sign nibble. If the number is negative, the sign nibble is changed to indicate a positive number. The number is pushed back on the stack, along with a real number 0 or 1 to indicate whether the sign changed.

```
ABSF 40 Bytes Checksum #A901h
( % → |%| %flag )
            CON(5)   =DOCODE    Code object prologue
            REL(5)   end        The length field — indicates the size of the code object
            GOSBVL   =POP1%     Pop a real number to A[W]
            ST=0     1          Clear status bit 1
            ?A=0     S          Test the sign nibble
            GOYES    Positive   If zero, the number is positivesign nibble
            A=0      S          Otherwise set the sign nibble to zero (positive)
            ST=1     1          Set status bit I to indicate sign change
Positive    GOSBVL   =PUSH%     Push the number back on the stack
            LC(5)    =%0        Prepare to push %0
            ?ST=0    1          Did the sign get changed?
            GOYES    PushIt     No, just push %0
            LC(5)    =%1        Yes, load address of %1
PushIt      A=C      A          Copy the address to A
            PC=(A)              Branch to the real number object
end
```

The code object ABSF1 does the same job, but returns `TRUE` or `FALSE`, using `PushT/FLoop`:

ABSF1 34.5 Bytes Checksum #9448h
(% → |%| %flag)

```
            CON(5)   =DOCODE       Code object prologue
            REL(5)   end           The length field — indicates the size of the code object
            GOSBVL   =POP1%        Pop a real number to A[W]
            ST=0     1             Clear status bit 1
            ?A=0     S             Test the sign nibble
            GOYES    Positive      If zero, the number is positivesign nibble
            A=0      S             Otherwise set the sign nibble to zero (positive)
            ST=1     1             Set status bit I to indicate sign change
Positive    GOSBVL   =PUSH%        Push the number back on the stack
            ?ST=0    1             Did the sign get changed?
            GOYES    PushIt        This test asserts the carry flag
PushIt      GOVLNG   =PushT/FLoop  Push the flag
end
```

11.6 Memory Utilities

When the RPL pointers are in the CPU, available memory can be calculated by subtracting B[A] (the end of the return stack) from the address in D1 (the first level of the data stack). If you're just pushing a pointer on the stack, just check that D[A] is non-zero.

11.6.1 Allocating Memory

Three entries are handy for allocating memory when a code object will be creating and returning a new object.

MAKE$	#05B79h
Creates a string object in TEMPOB with the specified number of characters. Generates an error exit if there isn't enough memory available to create the string and push it on the stack. Object *not* pushed on stack if error exit.	
Entry:	C[A]=desired number of characters, RPL pointers saved
Call with:	GOSBVL
Exit:	R0[A] → String, D0 → String body
Uses:	A[W], B[W], C[W], D[W], D0, D1, ST[0], ST[10]
Stack Levels:	3

MAKE$N	#05B7Dh
Creates a string object in TEMPOB with a length specified in nibbles. Generates an error exit if there isn't enough memory available to create the string and push it on the stack. Object *not* pushed on stack if error exit.	
Entry:	C[A]=string body length in nibbles, RPL pointers saved
Call with:	GOSBVL
Exit:	R0[A] → String, D0 → String body
Uses:	A[W], B[W], C[W], D[W], D0, D1, ST[0], ST[10]
Stack Levels:	3

GETTEMP	#039BEh
Allocates space in TEMPOB for an object	
Entry:	C[A]=number of nibbles to allocate, RPL pointers saved
Call with:	GOSBVL
Exit:	D0 → hole in TEMPOB
Uses:	A[W], B[W], C[W], D[W], D0, D1, ST[0], ST[10]
Stack Levels:	3

Notes:

- GETTEMP does not account for the room needed to push the object on the stack.

- If your code object is part of a library and if merged memory is in port 1 and the library is being executed out of a bank in port 2, the code object (or the secondary in which the code object is embedded) will be copied to TEMPOB and executed from there. In unusual circumstances, the object being executed can be deleted and overwritten by a garbage collection. It has been observed that when a garbage collection happens, no problems occur if the "ghost copy" of the object is not overwritten by a new object after garbage collection. You may wish to call `MAKE$N` with the assurance that a garbage collection will not happen. To do this, do a garbage collect first, or set status bit 10 and `GOSBVL ((=MAKE$N)+3)`. This technique is illustrated in MKSTR below.

Example: Create a String

MKSTR is a secondary containing a code object that creates a string of spaces given a bint. Note that this example has no type or range check code — a positive non-zero real number 1 is expected on the stack.

```
MKSTR 66 Bytes Checksum #E8F4h
( %characters → $ )
::                                 Code object prologue
  COERCE                           Convert real number character count into a bint
CODE
            GOSBVL  =POP#          Pop the bint into A[A]
            GOSBVL  =SAVPTR        Save the RPL pointers
            C=A     A              Copy character count into C[A]
            R1=C.F  A              Save character count in R1[A]
            C=C+C   A              Double C[A] to make string body size in nibbles
            ST=1    10             Flag garbage collected
            GOSBVL  ((=MAKE$N)+3)  Create the string object, error if not enough memory
            A=R1.F  A              Recover character count
            LCHEX   20             Character value for a space
WrtChr
            DAT0=C  B              Write space character
            D0=D0+  2              Advance the pointer
            A=A-1   A              Decrement the character count
            ?A#0    A              If there are more characters,
            GOYES   WrtChr           go write them
            GOSBVL  =GETPTR        Restore the RPL pointers to the CPU
            D1=D1-  5              Retard the stack pointer
            D=D-1   A              Decrement the available memory count
            A=R0.F  A              A[A]-string prologue
            DAT1=A  A              Write pointer to stack
            A=DAT0  A              Read pointer to next object in runstream
            D0=D0+  5              Advance return stack pointer
            PC=(A)                 Branch to next object in runstream
ENDCODE
;
```

11.6.2 Memory Move Utilities

The following memory utilities are available for moving memory.

`MOVEDOWN`	#0670Ch
Moves a block of memory from higher address to lower address	
Entry:	D0 → start of source, D1 → start of destination C[A]=number of nibbles to move RPL pointers saved
Call with:	`GOSBVL`
Exit:	D0 → end of source + 1, D1 → end of destination + 1, P=0
Uses:	A[W], C[A], D0, D1, P
Stack Levels:	0

`MOVEUP`	#066B9h
Moves a block of memory from lower address to higher address	
Entry:	D0 → end of source + 1, D1 → end of destination + 1 C[A]=number of nibbles to move RPL pointers saved
Call with:	`GOSBVL`
Exit:	D0 → start of source, D1 → start of destination, P=0
Uses:	A[W], C[A], D0, D1, P
Stack Levels:	0

`ECUSER`	#039EFh
Expand/contract an object in user memory	
Entry:	A[A] → insertion/deletion point C[A]=number of nibbles to expand/contract ST[5]=1 (expand) or ST[5]=0 (contract) D0 → Object prologue RPL pointers saved
Call with:	`GOSBVL`
Exit:	B[A] → start of new block or just above deleted block R0[A] = number of nibbles expanded/contracted Interrupts disabled (call SysRPL object `InitEnab` to re-enable) Garbage may be collected
Uses:	A, B, C, D, D0, D1, R0, R1,R2, P, ST[0], ST[2], ST[10]
Stack Levels:	4

Note that `ECUSER` *cannot* be called from a code object that's in TEMPOB or in USEROB, since TEMPOB may be adjusted during garbage collection, and USEROB will be altered. The safest places from which to use `ECUSER` are from port 0 or port 1.

Since `ECUSER` disables interrupts, you need to call `InitEnab` to restore interrupts.

`InitEnab`	#0970Ah
Enable interrupts after using ECUSER	
→	

Example: Expanding a String in UserOb

EXSTR (listed on the next page) illustrates the use of `ECUSER` by inserting the characters "AB" at the beginning of a string stored in a user variable. To try out EXSTR, do the following:

1. Download EXSTR to the HP 48.
2. Store it into a variable in port 0: `« 'EXSTR' RCL 0:EXSTR STO »`
3. Store a string into variable TEST, put its name on the stack, and execute EXSTR from port 0, then view the string:

```
« 'TEST' "12345" OVER STO 0:EXSTR EVAL TEST »
```

→"AB12345"

Note that you now have all the tools to write a small database application that stores data in Library Objects. Library objects are structured the same way as strings, except the prologue is different.

```
EXSTR 93.5 Bytes Checksum #F5CEh      (When stored in USEROB variable EXSTR)
( ID → )
::
  OLASTOWDOB! CK1NOLASTWD             Clear saved command name, one argument
  CK&DISPATCH1 idnt                   Require a global name object
  ::
    @ NOTcase SETNONEXTERR            Try to recall the variable, error if nonexistent
    DUPTYPECSTR? NOTcase SETTYPEERR   Generate error if variable does not contain a string
CODE
            A=DAT1   A                A[A]→string prologue
            D1=D1+   5                Pop the string
            D=D+1    A
            GOSBVL   =SAVPTR          Save RPL pointers
            D0=A                      D0→string prologue
            LC(5)    10               C[A] = size of prologue and length field
            A=A+C    A                A[A]→start of string body
            LC(5)    4                C[A]=number of nibbles to expand
            ST=1     5                Signal to expand
            GOSBVL   =ECUSER          Expand string object
            A=B A
            D1=A                      D1→expanded block start
            LCASC    \BA\             Load characters to write in C
            DAT1=A   4                Write new characters
            D1=D1-   5                D1→string length field
            A=DAT1   A                A[A]=old string length
            C=R0.F   A                C[A]=expansion size
            A=A+C    A                Add expansion size
            DAT1=A   A                Write new string length
            GOVLNG   =GETPTRLOOP
ENDCODE
  InitEnab                            Re-enable interrupts
  ;
;
```

11.6.3 Display Memory Addresses

The following techniques are useful for acquiring the addresses of display grobs in a version independent manner.

ADISP

Point D1 at the prologue of ABUFF

```
D1=(5)   (=addrADISP)+2
C=DAT1   A
D1=C
```

VDISP

Point D1 at the prologue of the currently displayed grob

```
D1=(5)   (=addrVDISP)+2
C=DAT1   A
D1=C
```

VIDSP2

Point D1 at the prologue of the menu grob

```
D1=(5)   (=addrVDISP2)+2
C=DAT1   A
D1=C
```

11.7 Reporting Errors

The assembly language analogue to the System-RPL object `ERRJMP` is the entry `Errjmp`. If you wish to generate an error using one of the built-in messages, load the message number in C[A] and go to `Errjmp`. There are two entries available for this:

Errjmp	#05023h
Stores the error number, restarts RPL at ERRJMP	
Entry:	A[A] = error#, RPL pointers in CPU
Call with:	`GOVLNG`
Exit:	To RPL

GPErrjmpC	#10F40h
Sets P=0, HEXMODE, restores RPL pointers, stores the error number, restarts RPL at ERRJMP	
Entry:	C[A] = error#, RPL pointers saved
Call with:	`GOVLNG`
Exit:	To RPL

The following code object pops a real number off the stack and generates a `Bad Argument Value` error if the number is negative.

```
ERR 30 Bytes Checksum #A915h
( % → )
            CON(5)   =DOCODE
            REL(5)   end
            GOSBVL   =POP1%        Pop a real number (sets DEC mode)
            SETHEX                 Reset HEX mode
            ?A=0     S             Test the sign nibble
            GOYES    Positive      If zero, just return to RPL
            LCHEX    00203         Otherwise load error message number
            GOVLNG   =GPErrjmpC       and generate the error
Positive
            GOVLNG   =GETPTRLOOP
```

11.8 Checking Batteries

If you're writing a code object that will be executing for a long time (like a game), you may wish to check the battery condition from time to time The entry `ChkLowBat` does this:

ChkLowBat	#325AAh
Checks for low battery	
Entry:	ST15=0 (interrupts disabled), RPL pointers saved
Call with:	`GOSBVL`
Exit:	CS: Low Battery and C[A]=LowBatErr#; CC: Battery OK
Uses:	A[A], B[A], C[A], D[A], D0, ST[7-0]
Stack Levels:	3

The following code object disables interrupts, checks the batteries using `ChkLowBat`, re-enables interrupts, and returns with a flag indicating the condition of the batteries.

```
CKBAT 28 Bytes Checksum #4297h
( → FLAG )
```

```
          CON(5)   =DOCODE
          REL(5)   end
          GOSBVL   =SAVPTR       Save the RPL pointers
          ST=0     15            Disable interrupts
          GOSBVL   =ChkLowBat    Check the batteries, assert the carry flag
          ST=1     15            Re-enable interrupts
          GOVLNG   =GPPushT/FLp  Push the flag based on carry
end
```

11.9 Warmstart & Coldstart

There may be times when you get into real trouble and a safe return to normal calculator execution is required. Perhaps you detect that memory isn't in good shape, something is missing, or a pointer is unreasonable. Three "last resort" options are available, listed in order of increasing severity:

- `GOVLNG =norecPWLseq ( #01FBDh )` Warmstarts without recording an entry in the warmstart log.
- `GOVLNG =Coldstart ( #01FD3h )` Branches to "Try To Recover Memory?" prompt.
- `GOVLNG =norecCSseq ( #01FDAh )` Unconditional memory clear (*total* coldstart).

The first option, a warmstart, may be used when you think TEMPOB is corrupt or other easily repairable system problems can be handled without risking the loss of USEROB. The second option may be required if you think USEROB is corrupt. It is impossible to imagine any use for the third "nuclear" option in a well-designed application. We discourage people who would use either the second or third option as a joke or prank — please confine your coding practices to those of responsible people.

11.10 Tone Generation

The entry `makebeep` can be used to generate steady tones at a specific frequency and duration, or you can generate your own sound effects by oscillating the beeper yourself.

11.10.1 Steady Tones

The entry `makebeep` respects the system beeper flag (-56) and checks the CPU speed to make as accurate a tone as possible.

`makebeep`	#017A6h
Generates a beep	
Entry:	C[A]=delay (msec) D[A]=frequency (Hz), RPL pointers saved
Call with:	`GOSBVL`
Exit:	*Interrupts ON (INTON)*
Uses:	A, B, C, D, R0, R1, R2, R3, D0, D1, P, Carry
Stack Levels:	1

```
TOOT 32 Bytes Checksum #21F1h
( → )
```

```
            CON(5)  =DOCODE
            REL(5)  end
            GOSBVL  =SAVPTR
            LC(5)   400
            D=C     A
            LC(5)   1000
            GOSBVL  =makebeep
            GOVLNG  =GETPTRLOOP
end
```

11.10.2 Rising and Falling Tones

The beeper is a piezoelectric element wired to bit 11 of the OUT register. You can click the beeper “on” by setting bit 11 and click it back “off” by clearing bit 11. *Remember to leave it off!* The example TONE shows how to generate sweeping tones by oscillating the beeper bit. As a courtesy to people who might use your code, please respect the status of the system beeper flag as shown below.

```
TONE 95.5 Bytes Checksum #534Ah
( → )
```

```
::
  56 TestSysFlag ?SEMI                      Exit if flag –56 is set
CODE
            GOSBVL   =SAVPTR                Save RPL pointers
            GOSUB    SweepUp                Generate rising sound
            LC(5)    8048                   Wait
Wait        C=C-1    A
            GONC     Wait
            GOSUB    SweepDn                Generate falling sound
            GOVLNG   =GETPTRLOOP            Restore RPL pointers and exit
*******************************************************************
* Subroutine SweepUp                                              *
*******************************************************************
SweepUp     LA(2)    130                    Starting tone (must be > ending tone)
UpLoop      LC(2)    3                      Intermediate delay
            GOSUB    Tone                   Generate the tone
            A=A-1    B                      Decrement tone value
            LC(2)    40                     Ending tone (must be < starting tone)
            ?A>C     B                      More tones to do?
            GOYES    UpLoop
            RTN
*******************************************************************
* Subroutine SweepDn                                              *
*******************************************************************
SweepDn     LA(2)    40                     Starting tone (must be < ending tone)
DnLoop      LC(2)    1                      Intermediate delay
            GOSUB    Tone                   Generate the tone
            A=A+1    B                      Increment the tone value
            LC(2)    130                    Ending tone (must be > starting tone)
            ?A<C     B                      More tones to do?
            GOYES    DnLoop
            RTN
*******************************************************************
* Subroutine Tone:  A[B] = Frequency C[B] = Intermediate delay      *
*******************************************************************
Tone        D=C      B                      Copy intermediate delay to D[B]
ToneLp      LCHEX    800                    Set bit 11
            OUT=C    B                      Click speaker ON
            C=A      B                      Copy tone value
Dec1        C=C-1    B                      Delay
            GONC     Dec1
            C=0      A                      Clear bit 11
            OUT=C                           Click speaker OFF
            C=A      B                      Copy tone value
Dec2        C=C-1    B                      Delay
            GONC     Dec2
            D=D-1    B                      Decrement tone length counter
            GONC     ToneLp                 Loop
            RTN
end
ENDCODE
;
```

11.11 Keyboard Scanning

The HP 48 keyboard is wired to the IN and OUT registers. During normal operation, the CPU scans the keyboard every millisecond and generates an interrupt when a key is pressed. Once the interrupt has been generated, the keyboard handler scans the keyboard to see which keys have been pressed. While a key is down, timer interrupts are scheduled to wake up the CPU every 1/16 of a second. This permits scans to see which key or keys are down, and lets the handler update the key buffer when a key is released. An application can scan the keyboard directly at full CPU speed, or shut down to save power between keystrokes. The former technique might be appropriate for a game where objects are moving; the latter might be better if the application is just waiting for user input.

To look for a particular key, set the appropriate bits of the OUT register, then AND the value from the IN register with an input mask. The table below shows the mask values for each key. For instance, the OUT mask for [CST] is 080 and the IN mask is 0008. The [ON] is mapped to bit 15 of IN only and generates a nonmaskable interrupt. To prevent the interrupt system from intercepting keys, you'll need to disable interrupts.

[A] 002/0010	[B] 100/0010	[C] 100/0008	[D] 100/0004	[E] 100/0002	[F] 100/0001
[MTH] 004/0010	[PRG] 080/0010	[CST] 080/0008	[VAR] 080/0004	[▲] 080/0022	[NXT] 080/0001
['] 001/0010	[STO] 040/0010	[EVAL] 040/0008	[◄] 040/0004	[▼] 040/0002	[►] 040/0001
[SIN] 008/0010	[COS] 020/0010	[TAN] 020/0008	$[\sqrt{x}]$ 020/0004	$[y^x]$ 020/0002	$[1/x]$ 020/0001
[ENTER] 010/0010		[+/–] 010/0008	[EEX] 010/0004	[DEL] 010/0002	[←] 010/0001

[α] 008/0020	[7] 008/0008	[8] 008/0004	[9] 008/0002	[÷] 008/0001
[↰] 004/0020	[4] 004/0008	[5] 004/0004	[6] 004/0002	[X] 004/0001
[↱] 002/0020	[1] 002/0008	[2] 002/0004	[3] 002/0002	[-] 002/0001
[ON] /8000	[0] 001/0008	[.] 001/0004	[SPC] 001/0002	[+] 001/0001

The following subroutine tests the keyboard and returns with carry set if [▼] is down. Note that the `C=IN` instruction *must* be executed from an even address. To do this reliably, call `CINRTN`, which just does `C=IN` and returns.

```
        LCHEX    00040
        OUT=C
        GOSBVL   =CINRTN
        LAHEX    00002
        C=A&C    A
        ?A#0     A
        RTNYES
        RTN
```

11.11.1 Managing Interrupts

If you're going to look for keys yourself, it's best to disable keyboard scanning. This frees up CPU time for your application and avoids unwanted keystrokes wandering into the key buffer. There are three methods of disabling interrupts, listed in order of decreasing severity:

- Call the entry `DisableIntr` to disable all interrupts, and `AllowIntr` to enable interrupts. This shuts off all I/O, and carries the risk that if your code goes astray only a "paperclip reset" is possible (pushing a paperclip in the hole under the upper-right rubber foot).

`DisableIntr`	#01115h
Disable interrupts	
Entry:	Interrupts enabled
Call with:	`GOSBVL`
Exit:	Interrupts disabled
Uses:	C[A], Carry
Stack Levels:	1

`AllowIntr`	#010E5h
Re-enable interrupts	
Entry:	Interrupts disabled
Call with:	`GOSBVL`
Exit:	Interrupts enabled
Uses:	C[A] , Carry
Stack Levels :	1

- Clear bit 15 of the status register. This shuts off all I/O, and carries the risk that if your code goes astray only a "paperclip reset" is possible.
- Execute the `INTOFF` instruction. This prevents *only* keyboard interrupts except for [ON], which always generates an interrupt. This has the advantage that you can use [ON] — [C] to recover from code bugs. The disadvantage is that the [ON] key can't be detected reliably and will be processed by the interrupt system. Note that makebeep, the ticking clock display, or alarms can lead to an INTON instruction being executed.

11.11.2 Rapid Keyboard Scans

The example KEY1 scans the keyboard at full speed, exiting only when either [ON] or [F] have been pressed and released.

```
KEY1 50.5 Bytes Checksum #FD27h
( → )
```

```
        CON(5)   =DOCODE
        REL(5)   end
        ST=0     15           Turn off interrupts
        LAHEX    08001        Input mask for [F] and [ON]
Top     LCHEX    00100        Output mask for [F]
        OUT=C                 Set keyboard lines to look for [F]
        GOSBVL   =CINRTN      Read back the keyboard lines
        C=A&C    A            Mask off lines for [F] and [ON]
        ?C=0     A            Were either of our keys pressed?
        GOYES    Top          No, go scan again
StillDn LCHEX    001FF        Output mask for all rows
        OUT=C
        GOSBVL   =CINRTN      Read back keyboard state
        ?C#0     A            Are there still keys down?
        GOYES    StillDn      Yes, go scan again
        ST=1     15           No, re-enable interrupts
        A=DAT0   A            Back to RPL
        D0=D0+   5
        PC=(A)
end
```

The example KEY2 scans the keyboard until [ON] is pressed. During the scan [A] turns on a small line in the display, and [B] turns the line off.

```
KEY2 125.5 Bytes Checksum #57E1h
( → )
```

```
                CON(5)   =DOCODE
                REL(5)   end
                GOSBVL   =SAVPTR                Save RPL pointers
                D1=(5)   (=addrADISP) +2        Point D1 at the address of ABUFF's address
                A=DAT1   A                      Load the ABUFF address's address into A[A]
                D1=A                            Copy to D1
                A=DAT1   A                      Read the address of ABUFF
                D1=A                            D1→ABUFF prologue
                D1=D1+   15                     Skip past prologue, length, dimensions
                D1=D1+   5                      D1→First nibble of ABUFF data
                ST=0     15                     Turn off interrupts
                GOSUB    StillDn?               Wait for no keys pressed

Top             LCHEX    001FF                  Load mask for all rows
                OUT=C                           Set keyboard lines
                GOSBVL   =CINRTN                Read keyboard state
                ?C=0     A                      Any keys pressed?
                GOYES    Top                    No, go wait for a key

                LCHEX    002                    Look for [A] — first load row mask
                OUT=C
                GOSBVL   =CINRTN
                LAHEX    010                    Load column mask for [A]
                C=A&C    X
                ?C=0     X                      Did we get [A]?
                GOYES    TryB                   No, go test for [B]
                GOSUB    StillDn?               Yes, wait for key up
                LAHEX    FFFFF                  Load pattern to write to display
                DAT1=A   A                      Write pattern
                GOTO     Top                    Go back for another key

TryB            LCHEX    100                    Load row mask for [B]
                OUT=C
                GOSBVL   =CINRTN
                LAHEX    010                    Load column mask for [B]
                C=A&C    X
                ?C=0     X                      Did we get [B]?
                GOYES    TryON                  No, go test for [ON]
                GOSUB    StillDn?               Yes, wait for key up
                A=0      A                      Load pattern to write to display
                DAT1=A   A                      Write pattern
                GOTO     Top                    Go back for another key

TryON           LAHEX    08000                  Load mask for [ON]
                C=A&C    A
                ?C#0     A                      Did we get [ON]?
                GOYES    GotON                  Yes, go quit
                GOTO     Top                    No, go look for another key

GotON           GOSUB    StillDn?               Wait for key up
                GOTO     Done                   Go finish

StillDn?        LCHEX    001FF                  Load row mask for all keys
                OUT=C
                GOSBVL   =CINRTN
                ?C#0     A                      Was a key down?
                GOYES    StillDn?               Yes, loop until no keys are down
```

```
             RTN                        No, return

Done         ST=1    15                 Re-enable interrupts
             GOVLNG  =GETPTRLOOP        Back to RPL
end
```

11.11.3 Low Power Keyboard Scans

You can save power by putting the calculator into a low power state between keystrokes. We'll describe some of the basic pieces, then put them all together in the example KEY3.

The Basic Timer Loop. In the basic low power loop a timer is set to wake the calculator up after a small interval, then the SHUTDN instruction is executed, putting the calculator in a low power state. The calculator can wake up for several reasons, including a timer expiring or a key being pressed. The technique we show here ignores other reasons for wakeup. When the calculator wakes up the keyboard is scanned and if no keys are down the timer is reset and the calculator goes to sleep again.

```
LiteSlp      D1=(5)  =TIMERCTRL.1       Set timer 1 to wake up CPU
             LC(1)   4
             DAT1=C  P
             D1=(2)  =TIMER1            Set a 5/16 second delay
             LC(1)   5
             DAT1=C  P
             LCHEX   1FF                Preload the keyboard row lines
             OUT=C
Wait         SHUTDN                     WAIT FOR INTERRUPTS
             LC(3)   1FF                Load the row lines
             OUT=C
             GOSBVL  =CINRTN            Read the column lines
             LAHEX   0803F              Mask for all column lines
             A=A&C   A
             ?A#0    A                  Was a key pressed?
             GOYES   GetKey             Yes, go see which one(s) are down
             D1=(2)  =TIMERCTRL.1       No, so look at timer control
             C=DAT1  X                  Read timer status
             ?CBIT=0 3                  Was timer expired?
             GOYES   Wait               No, go back to sleep
             GOSUB   Blink              Yes, blink the cursor
             GOTO    LiteSlp            Then go back to sleep
```

Keyboard Debounce. The entry `Debounce` scans the keyboard until it has been stable for at least one timer tick:

`Debounce`	#009A5h
Scan the keyboard until stable, return bitmap of pressed keys	
Entry:	*Interrupts disabled*
Call with:	`GOSBVL`
Exit:	A[12-0]=Key bitmap
Uses:	A, B, C, D[A], D0, D1, P, SB, Carry
Stack Levels:	0

The bits returned in A[12-0] encode keys as shown in the table below. Note that more than one key may be down.

Nibble	Bit 3	Bit 2	Bit 1	Bit 0
12				[B]
11	[C]	[D]	[E]	[F]
10	[PRG]	[CST]	[VAR]	▲
9	[NXT]	[STO]	[EVAL]	◄
8	▼	►	[COS]	[TAN]
7	[$\sqrt{x}$]	[y^x]	[$^1/_x$]	[ENTER]
6	[+/−]	[EEX]	[DEL]	⇐
5	α	[SIN]	[7]	[8]
4	[9]	[÷]	↰	[MTH]
3	[4]	[5]	[6]	[X]
2	↱	[A]	[1]	[2]
1	[3]	[-]	[']	[0]
0	[.]	[SPC]	[+]	[ON]

The Key Bitmap. After obtaining the bitmap, you can either load a 13 nibble mask to look for one or more specific keys, or you can generate a number corresponding to the key that was down. In the latter case, you may wish to ensure that just one key is down. The following code fragment (not used in the KEY3 example) returns the number of keys pressed in C[B] given a key bitmap in B[W]:

```
Entry:                                   B[W] = key bitmap
Call with:                               GOSUB CountKeys
Exit:                                    C[B] = # of keys down, Carry set

CountKeys  C=0     B                     Clear the key counter
AnySet?    ?B=0    W                     Are all bits clear?
           RTNYES                        Return if so
TstNib     ?B#0    P                     Is the least significant nibble clear?
           GOYES   TstBit                No, go check the bits in that nibble
           BSR     W                     Yes, shift in next nibble
           GONC    AnySet?               Go see if there's more to test
*
TstBit     B=B+B   P                     Shift nibble left, set carry if high bit was set
           GONC    TstBit                If the high bit was clear, shift again
           C=C+1   B                     Increment key counter
           GONC    TstNib                Go see if more bits are set in this nibble
```

The following code fragment returns in B[A] the option-base-1 number of the least significant set bit in a keymap in A[W]. The key number ranges from 1 ([ON]) to 49 ([B]).

```
Entry:                                   A[W] = key bitmap with at least one bit set
Call with:                               GOSUB KeyNum
Exit:                                    B[A] = key number

KeyNum     B=0      A                    Clear the key number
NextNib    ?A#0     P                    Is the least significant nibble clear?
           GOYES    TestBits             No, go find which bit is set
           B=B+CON  B,4                  Yes, add four to the key number,
           ASR      W                    shift the next nibble in,
           GONC     NextNib              (BET) and go test the next nibble

TestBits   B=B+1    B                    Increment the key number
           SB=0                          Clear the sticky bit
           ASRB.F   P                    Shift off a bit
           ?SB=0                         Was it set?
           GOYES    TestBits             No, go test the next bit
           RTN                           Yes, return with key number in B[A]
```

Putting it All Together. The example KEY3 blinks a cursor line in the display while waiting for a key in light sleep. When a key is pressed, the keycode is returned to the stack as a real number.

```
KEY3 201.5 Bytes Checksum #28B2h
```

(→ %keycode)

```
::
  CLEARVDISP
CODE
            GOSBVL   =SAVPTR
            D1=(5)   (=addrADISP)+2
            A=DAT1   A
            D1=A
            A=DAT1   A
            LC(5)    20
            A=A+C    A
            R1=A
            GOSUB    WaitKeyUp
            GOSBVL   =DisableIntr
            GOSUB    BusyOff
            ST=0     1
LiteSlp     D1=(5)   =TIMERCTRL.1         Set timer 1 to wake up CPU
            LC(1)    4
            DAT1=C   P
            D1=(2)   =TIMER1              Set a 5/16 second delay
            LC(1)    5
            DAT1=C   P
            LCHEX    1FF                  Preload the keyboard row lines
            OUT=C
Wait        SHUTDN                        WAIT FOR INTERUPTS
            LC(3)    1FF                  Load the row lines
            OUT=C
            GOSBVL   =CINRTN              Read the column lines
            LAHEX    0803F                Mask for all column lines
            A=A&C    A
            ?A#0     A                    Was a key pressed?
            GOYES    GetKey               Yes, go see which one(s) are down
            D1=(2)   =TIMERCTRL.1         No, so look at timer control
            C=DAT1   X                    Read timer status
            ?CBIT=0  3                    Was timer expired?
            GOYES    Wait                 No, go back to sleep
            GOSUB    Blink                Yes, blink the cursor
            GOTO     LiteSlp              Then go back to sleep

GetKey      GOSBVL   =Debounce            Debounce the keyboard, create bitmap in A
            ?A#0     W                    Was a key pressed?
            GOYES    GotKey               Yes, go create a keycode
            GOTO     LiteSlp              No, go wait again

GotKey      GOSUB    KeyNum               Get the key number
            A=0      A                    Clear A[A]
            A=B      B                    Copy to A,
            R0=A.F   A                    Save in R0 for PUSH#
            GOSUB    WaitKeyUp            Wait for the key to be released
            GOSBVL   =AllowIntr           Re-enable interrupts
            GOSUB    BusyOn               Turn on the busy annunciator
            GOSBVL   =PUSH#               Push the key number, restore RPL pointers
            LC(5)    =UNCOERCE            Return to RPL, executing UNCOERCE
            A=C      A
            PC=(A)

Subroutine to wait for keys to be released:
```

```
WaitKeyUp   LCHEX     1FF             Set row lines
            OUT=C
            GOSBVL    =CINRTN         Read column lines
            LAHEX     0803F           Mask for column lines
            A=A&C     A
            ?A#0      A               Were any keys down?
            GOYES     WaitKeyUp       Yes, go scan again
            RTN                       No, return
```

Subroutine to blink cursor:

```
Blink       C=0       A               Clear C[A] to clear cursor
            ?ST=0     1               Was cursor off?
            GOYES     TurnOn          Yes, go turn it on
            ST=0      1               Turn off cursor status bit
            GONC      Write           Go write the cursor
TurnOn
            ST=1      1               Turn on cursor status bit
            C=C-1     A               Set C[A] to FFFFF
Write       A=R1.F    A               Recover pointer to display
            D1=A                      Copy to D1
            DAT1=C    A               Write cursor
            RTN
```

Subroutine to turn off busy annunciator:

```
BusyOff     D0=(5)    (=ANNCTRL)+1    Point at the annunciator nibble
            C=DAT0    P               Read nibble
            CBIT=0    0               Clear annunciator bit
WrtRtn      DAT0=C    P               Write nibble back
            RTN
```

Subroutine to turn on the busy annunciator:

```
BusyOn      D0=(5)    (=ANNCTRL)+1    Point at the annunciator nibble
            C=DAT0    P               Read nibble
            CBIT=1    0               Set annunciator bit
            DAT0=C    P               Write nibble back
            RTN
```

Subroutine to calculate the key number:

```
KeyNum      B=0       A               Clear the key number
NextNib     ?A#0      P               Is the least significant nibble clear?
            GOYES     TestBits        No, go find which bit is set
            B=B+CON   B,4             Yes, add four to the key number,
            ASR       W               shift the next nibble in,
            GONC      NextNib         (BET) and go test the next nibble
*
TestBits    B=B+1     B               Increment the key number
            SB=0                      Clear the sticky bit
            ASRB.F    P               Shift off a bit
            ?SB=0                     Was it set?
            GOYES     TestBits        No, go test the next bit
            RTN                       Yes, return with key number in B[A]
ENDCODE
```

```
;
```

Processing Keycodes. Once you have a keycode from the KeyNum subroutine, there are several ways to branch to the corresponding code. The first is best if your application defines only a few keys — just compare individual key codes. The second is best if your application defines many keys. Both examples assume a key number in B[A], and that the return to get another key is at the label `LiteSlp`.

The first example looks for [ENTER], [◄] , and [►] :

```
           LC(2)    29                  Key number for [ENTER]
           ?B#C     B
           GOYES    TryLeft
           GOTO     DoEnter
*
TryLeft    LC(2)    37                  Key number for [◄]
           ?B#C     B
           GOYES    TryRight
           GOTO     DoLeft
*
TryRight   LC(2)    35                  Key number for [►]
           ?B=C     B
           GOYES    GoDoRight
           GOTO     LiteSlp             Go for another key if not [►]
GoDoRight  GOTO     DoRight
*
DoEnter    Process [ENTER]
           GOTO     LiteSlp
*
DoLeft     Process [◄]
           GOTO     LiteSlp
*
DoRight    Process [►]
           GOTO     LiteSlp
```

The second example uses a table of 3-nibble offsets to the key subroutines. (Note that if your application is very large, you may need to use 4-nibble offsets.) The trick is to GOSUB around the table, which puts the table's starting address on the return stack.

Note that the references to the subroutines *must be forward references*, meaning that the key subroutines must come after the table. If the subroutine was before the table, each table entry would have to be 5 nibbles to make the address calculation correct.

```
           GOSUB    SendKey
           REL(3)   DoON                Pointer for [ON]
           REL(3)   DoPlus              Pointer for [+] (1)
           ...
           REL(3)   LiteSlp             Pointer for undefined key
           ...
           REL(3)   DoB                 Pointer for [B] (49)
SendKey    B=B-1    A                   Make option base 0 key number
           C=RSTK                       Get address of key table
           C=C+B    A                   Add keynumber*3 to start of table
           C=C+B    A
           C=C+B    A
           D0=C                         D0→key entry
           A=0      A
           A=DAT0   X                   Read offset to key routine
           C=A+C    A                   Add offset to table entry location
           PC=C                         Branch to key routine
```

The example KEY4 (on the disk, but not listed here) uses this technique.

11.12 The RVIEW Debugging Tool

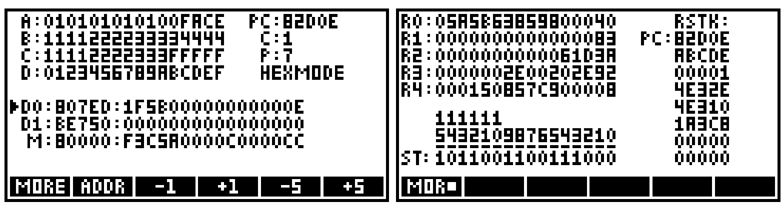
The subroutine RVIEW (Register VIEWer) has been provided to provide an additional example of various techniques for writing code in assembly language and as a simple debugging aid that you can use as you develop your programs. RVIEW is small, just a few thousand bytes in size, so you don't have to allocate a lot of memory to use it. RVIEW is in the RVIEW directory on the disk.

RVIEW will run on either S or G series calculators, but has three restrictions:

- The stack grob ABUFF must be full height — 64 rows. Note that by default ABUFF is 56 rows high, so you may need to enlarge ABUFF (see *Graphics* on page 113).
- RVIEW is self-modifying, so you cannot run RVIEW from a write-protected card.
- RVIEW consumes three stack levels, so be sure they're available.

11.12.1 The RVIEW User Interface

When RVIEW is executed, it saves the state of the CPU, displays the CPU register contents and windows into memory, then restores the CPU and ABUFF to their original state upon exit. RVIEW has two screens, selected with the |MORE| softkey:



In the first screen, the pointer arrow ▶ refers to the active memory window — D0, D1, or M.

While RVIEW is active, the following keys are active:

[On] [◄]	Quits RVIEW
[▲] [▼]	Moves the pointer arrow between the three memory windows
[◄] [►]	Increments or decrements the address of the active memory window
|MORE|	Switches the display between the two screens
|ADDR|	Lets you type a new address for the active memory window
| -1 |	Decrements the address of the active memory window
| +1 |	Increments the address of the active memory window
| -5 |	Subtracts 5 from the address of the active memory window
| +5 |	Adds 5 to the address of the active memory window

From the first screen, you can press [NXT] to display additional menu labels for address modification:

|-100|	Subtracts #100h from the address of the active memory window
|+100|	Adds #100h to the address of the active memory window
|-1000|	Subtracts #1000h from the address of the active memory window
|+1000|	Adds #1000h to the address of the active memory window

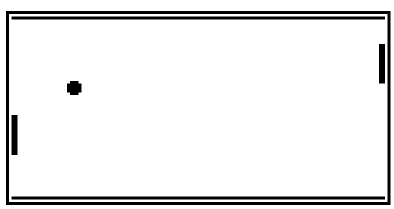
11.12.2 Using RVIEW

To use RVIEW in your code, just add the RVIEW source to your code and call RVIEW with a GOSUB. For instance, if you were going to include RVIEW in the SWP example to see the stack before and after the swap operation, the code would look like this:

```
            NIBASC   /HPHP48-A/          This is a download header for binary transfer to the
                                         HP 48
            CON(5)   =DOCODE             This is the prologue for a code object
            REL(5)   end                 The length field — indicates the size of the code object
            GOSUB    RVIEW
            A=DAT1   A                   Copy the stack level 1 pointer to A[A]
            D1=D1+   5                   Advance D1 to stack level 2
            C=DAT1   A                   Copy the stack level 2 pointer to C[A]
            DAT1=A   A                   Replace stack level 2 with the original stack level 1
                                         pointer
            D1=D1-   5                   Move D1 back to stack level 1
            DAT1=C   A                   Replace stack level 1 with the original stack level 2
                                         pointer
            GOSUB    RVIEW
            A=DAT0   A                   Read the pointer to the next RPL object to be executed
            D0=D0+   5                   Advance the instruction pointer
            PC=(A)                       Branch to the next instruction
*
RVIEW
            RVIEW source code here
end
```

11.13 The PONG Game

The directory PONG on the disk contains an HP 48 implementation of the classic PONG game, implemented as a compiled secondary including the game as a code object. To run the game transfer the file PONG to your HP 48 and execute PONG.



When PONG is running, the following keys are active:

[ON] [←] Quits PONG

[A] Moves the left player's paddle up

[G] Moves the left player's paddle down

[F] Moves the right player's paddle up

[L] Moves the right player's paddle down

The file MAKEPONG.BAT is a DOS batch file that will make the game based on the files PONG.S and PONG.M.

We hope this will inspire some more games!

Appendix A

Messages

Hex	Dec	General Messages
001	1	Insufficient Memory
002	2	Directory Recursion
003	3	Undefined Local Name
004	4	Undefined XLIB Name
005	5	Memory Clear
006	6	Power Lost
007	7	Warning:
008	8	Invalid Card Data
009	9	Object In Use
00A	10	Port Not Available
00B	11	No Room in Port
00C	12	Object Not in Port
00D	13	Recovering Memory
00E	14	Try To Recover Memory?
00F	15	Replace RAM, Press ON
010	16	No Mem To Config All
101	257	No Room to Save Stack
102	258	Can't Edit Null Char.
103	259	Invalid User Function
104	260	No Current Equation
106	262	Invalid Syntax

Hex	Dec	Object Types
107	263	Real Number
108	264	Complex Number
109	265	String
10A	266	Real Array
10B	267	Complex Array
10C	268	List
10D	269	Global Name
10E	270	Local Name
10F	271	Program
110	272	Algebraic
111	273	Binary Integer
112	274	Graphic
113	275	Tagged
114	276	Unit
115	277	XLIB Name
116	278	Directory
117	279	Library
118	280	Backup
119	281	Function
11A	282	Command
11B	283	System Binary
11C	284	Long Real
11D	285	Long Complex
11E	286	Linked Array
11F	287	Character
120	288	Code
121	289	Library Data
122	290	External

Hex	Dec	General Messages
123	291	Null message
124	292	LAST STACK Disabled
125	293	LAST CMD Disabled
126	294	HALT Not Allowed
127	295	Array
128	296	Wrong Argument Count
129	297	Circular Reference
12A	298	Directory Not Allowed
12B	299	Non-Empty Directory
12C	300	Invalid Definition
12D	301	Missing Library
12E	302	Invalid PPAR
12F	303	Non-Real Result
130	304	Unable to Isolate

Hex	Dec	Low Memory
131	305	No Room to Show Stack
132	306	Warning
133	307	Error:
134	308	Purge?
135	309	Out of Memory
136	310	Stack
137	311	Last Stack
138	312	Last Commands
139	313	Key Assignments
13A	314	Alarms
13B	315	Last Arguments
13C	316	Name Conflict
13D	317	Command Line

Hex	Dec	Stack Operations
201	513	Too Few Arguments
202	514	Bad Argument Type
203	515	Bad Argument Value
204	516	Undefined Name
205	517	LASTARG Disabled

Hex	Dec	EquationWriter
206	518	Incomplete Subexpression
207	519	Implicit ( ) off
208	520	Implicit ( ) on

Hex	Dec	Floating Point Errors
301	769	Positive Underflow
302	770	Negative Underflow
303	771	Overflow
304	772	Undefined Result
305	773	Infinite Result

Hex	Dec	Array
501	1281	Invalid Dimension
502	1282	Invalid Array Element
503	1283	Deleting Row
504	1284	Deleting Column
505	1285	Inserting Row
506	1286	Inserting Column

Hex	Dec	Statistics
601	1537	Invalid $\sum$ Data
602	1538	Nonexistent $\sum$DAT
603	1539	Insufficient $\sum$ Data
604	1540	Invalid $\sum$PAR
605	1541	Invalid $\sum$ Data LN(Neg)
606	1542	Invalid $\sum$ Data LN(0)

Hex	Dec	Plot, Solve, Stat
607	1543	Invalid EQ
608	1544	Current equation:
609	1545	No current equation.
60A	1546	Enter eqn, press NEW
60B	1547	Name the equation, press ENTER
60C	1548	Select plot type
60D	1549	Empty catalog
60E	1550	undefined
60F	1551	No stat data to plot
610	1552	Autoscaling
611	1553	Solving for
612	1554	No current data. Enter
613	1555	data point, press $\sum$+
614	1556	Select a model

Hex	Dec	Alarms
615	1557	No alarms pending.
616	1558	Press ALRM to create
617	1559	Next alarm:
618	1560	Past due alarm:
619	1561	Acknowledged
61A	1562	Enter alarm, press SET
61B	1563	Select repeat interval

Hex	Dec	I/O, Plot, Solve, Stat
61C	1564	I/O setup menu
61D	1565	Plot type:
61E	1566	" "
61F	1567	(OFF SCREEN)
620	1568	Invalid PTYPE
621	1569	Name the stat data, press ENTER
622	1570	Enter value (zoom out if >1), press ENTER

Hex	Dec	I/O, Plot, Solve, Stat
623	1571	Copied to stack
624	1572	x axis zoom w/AUTO.
625	1573	x axis zoom.
626	1574	y axis zoom.
627	1575	x and y-axis zoom.
628	1576	IR/wire:
629	1577	ASCII/binary:
62A	1578	baud:
62B	1579	parity:
62C	1580	checksum type:
62D	1581	translate code:
62E	1582	Enter matrix, then NEW
A01	2561	Bad Guess(es)
A02	2562	Constant?
A03	2563	Interrupted
A04	2564	Root
A05	2565	Sign Reversal
A06	2566	Extremum
A07	2567	Left
A08	2568	Right
A09	2569	Expr

Hex	Dec	Unit Management
B01	2817	Invalid Unit
B02	2818	Inconsistent Units

Hex	Dec	I/O and Printing
C01	3073	Bad Packet Block Check
C02	3074	Timeout
C03	3075	Receive Error
C04	3076	Receive Buffer Overrun
C05	3077	Parity Error
C06	3078	Transfer Failed
C07	3079	Protocol Error
C08	3080	Invalid Server Cmd.
C09	3081	Port Closed
C0A	3082	Connecting
C0B	3083	Retry #
C0C	3084	Awaiting Server Cmd.
C0D	3085	Sending
C0E	3086	Receiving
C0F	3087	Object Discarded
C10	3088	Packet #
C11	3089	Processing Command
C12	3090	Invalid IOPAR
C13	3091	Invalid PRTPAR
C14	3092	Low Battery
C15	3093	Empty Stack
C16	3094	Row
C17	3095	Invalid Name

Hex	Dec	Time
D01	3329	Invalid Date
D02	3330	Invalid Time
D03	3331	Invalid Repeat
D04	3332	Nonexistent Alarm

Hex	Dec	Polynomial Root Finder
0001	49153	Unable to find root

Hex	Dec	Multiple Equation Solver
E401	58369	Invalid Mpar
E402	58370	Single Equation
E403	58371	EQ Invalid for MINIT
E404	58372	Too Many Unknowns
E405	58373	All Variables Known
E406	58374	Illegal During MROOT
E407	58375	Solving for
E408	58376	Searching

Start	End	Unlisted Message Numbers
B901	B99B	Miscellaneous
BA01	BA43	I/O operations
BB01	BB3F	Statistics
BC01	BC3B	Time system
BD01	BD27	Symbolic operations
BE01	BE77	Plotting
BF01	BF56	Solver
E101	E129	Constants Library
E301	E304	Equation Library
E601	E60D	TVM Library
E701	E706	Minehunt game

Appendix B

Character Codes

DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR
0	00	▪	32	20		64	40	@	96	60	`
1	01	▪	33	21	!	65	41	A	97	61	a
2	02	▪	34	22	"	66	42	B	98	62	b
3	03	▪	35	23	#	67	43	C	99	63	c
4	04	▪	36	24	$	68	44	D	100	64	d
5	05	▪	37	25	%	69	45	E	101	65	e
6	06	▪	38	26	&	70	46	F	102	66	f
7	07	▪	39	27	'	71	47	G	103	67	g
8	08	▪	40	28	(	72	48	H	104	68	h
9	09	▪	41	29	)	73	49	I	105	69	i
10	0A	▪	42	2A	*	74	4A	J	106	6A	j
11	0B	▪	43	2B	+	75	4B	K	107	6B	k
12	0C	▪	44	2C	,	76	4C	L	108	6C	l
13	0D	▪	45	2D	-	77	4D	M	109	6D	m
14	0E	▪	46	2E	.	78	4E	N	110	6E	n
15	0F	▪	47	2F	/	79	4F	O	111	6F	o
16	10	▪	48	30	0	80	50	P	112	70	p
17	11	▪	49	31	1	81	51	Q	113	71	q
18	12	▪	50	32	2	82	52	R	114	72	r
19	13	▪	51	33	3	83	53	S	115	73	s
20	14	▪	52	34	4	84	54	T	116	74	t
21	15	▪	53	35	5	85	55	U	117	75	u
22	16	▪	54	36	6	86	56	V	118	76	v
23	17	▪	55	37	7	87	57	W	119	77	w
24	18	▪	56	38	8	88	58	X	120	78	x
25	19	▪	57	39	9	89	59	Y	121	79	y
26	1A	▪	58	3A	:	90	5A	Z	122	7A	z
27	1B	▪	59	3B	;	91	5B	[	123	7B	{
28	1C	▪	60	3C	<	92	5C	\	124	7C	|
29	1D	▪	61	3D	=	93	5D	]	125	7D	}
30	1E	▪	62	3E	>	94	5E	^	126	7E	~
31	1F	…	63	3F	?	95	5F	_	127	7F	▒

DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR
128	80	∡	160	A0		192	C0	À	224	E0	à
129	81	⧖	161	A1	¡	193	C1	Á	225	E1	á
130	82	∇	162	A2	¢	194	C2	Â	226	E2	â
131	83	√	163	A3	£	195	C3	Ã	227	E3	ã
132	84	∫	164	A4	¤	196	C4	Ä	228	E4	ä
133	85	Σ	165	A5	¥	197	C5	Å	229	E5	å
134	86	▶	166	A6	¦	198	C6	Æ	230	E6	æ
135	87	π	167	A7	§	199	C7	Ç	231	E7	ç
136	88	∂	168	A8	¨	200	C8	È	232	E8	è
137	89	≤	169	A9	©	201	C9	É	233	E9	é
138	8A	≥	170	AA	ª	202	CA	Ê	234	EA	ê
139	8B	≠	171	AB	«	203	CB	Ë	235	EB	ë
140	8C	∝	172	AC	¬	204	CC	Ì	236	EC	ì
141	8D	→	173	AD	-	205	CD	Í	237	ED	í
142	8E	←	174	AE	®	206	CE	Î	238	EE	î
143	8F	↓	175	AF	¯	207	CF	Ï	239	EF	ï
144	90	↑	176	B0	°	208	D0	Ð	240	F0	ð
145	91	γ	177	B1	±	209	D1	Ñ	241	F1	ñ
146	92	δ	178	B2	²	210	D2	Ò	242	F2	ò
147	93	∈	179	B3	³	211	D3	Ó	243	F3	ó
148	94	η	180	B4	´	212	D4	Ô	244	F4	ô
149	95	θ	181	B5	µ	213	D5	Õ	245	F5	õ
150	96	λ	182	B6	¶	214	D6	Ö	246	F6	ö
151	97	ρ	183	B7	·	215	D7	×	247	F7	÷
152	98	σ	184	B8	¸	216	D8	Ø	248	F8	ø
153	99	τ	185	B9	¹	217	D9	Ù	249	F9	ù
154	9A	ω	186	BA	º	218	DA	Ú	250	FA	ú
155	9B	Δ	187	BB	»	219	DB	Û	251	FB	û
156	9C	Π	188	BC	¼	220	DC	Ü	252	FC	ü
157	9D	Ω	189	BD	½	221	DD	Ý	253	FD	ý
158	9E	■	190	BE	¾	222	DE	Þ	254	FE	þ
159	9F	∞	191	BF	¿	223	DF	ß	255	FF	ÿ

Appendix C

Flags

User flags are numbered 1 through 64. System flags are numbered from –1 through –64. By convention, application developers are encouraged to restrict their use of user flags to the range 31–64. All flags are clear by default, except for the wordsize (flags –5 to –10).

Flag	Description	Clear	Set	Default
Symbolic Math				
–1	Principal Solution	General solutions	Principal solutions	Clear
–2	Symbolic Constants	Symbolic form	Numeric form	Clear
–3	Numeric Results	Symbolic results	Numeric results	Clear
–4	Not used			
Binary Integer Math				
–5	Binary integer wordsize n + 1: $0 \le n \le 63$			64
–10	Flag –10 is the most significant bit			
	Base	**–11**	**–12**	DEC
–11	DEC	Clear	Clear	
and	BIN	Clear	Set	
–12	OCT	Set	Clear	
	HEX	Set	Set	
–13	Not used			
Finance				
–14	TVM Payment Mode	End of Period	Beginning of Period	End
Coordinate System		**–15**	**–16**	Rect.
–15	Rectangular	Clear	Clear	
and	Cylindrical Polar	Clear	Set	
–16	Spherical Polar	Set	Set	
Trigonometric Mode		**–17**	**–18**	Degrees
–17	Degrees	Clear	Clear	
and	Radians	Set	Clear	
–18	Grads	Clear	Set	
Math Exception				
–19	Vector/complex	Vector	Complex	Vector
–20	Underflow Exception	Return 0, set flag –23 or –24	Error	Clear
–21	Overflow Exception	Return ±MAXR, set flag –25	Error	Clear
–22	Infinite Result	Error	Return ±MAXR, set flag –26	Error
–23	Pos. Underflow Indicator	No Exception	Exception	Clear
–24	Neg. Underflow Indicator	No Exception	Exception	Clear
–25	Overflow Indicator	No Exception	Exception	Clear
–26	Infinite Result Indicator	No Exception	Exception	Clear
–27	Symbolic Decompilation	'X+Y*i'->'(X,Y)'	'X+Y*i'->'X+Y*i'	Clear
Plotting and Graphics				
–28	Plotting Multiple Functions	Plotted serially	Plotted simultaneously	Clear
–29	Trace mode	Trace off	Trace on	Off
–30	Not used			
–31	Curve Filling	Filling enabled	Filling disabled	Enabled
–32	Graphics Cursor	Visible light bkgnd	Visible dark bkgnd	Light

Flag	Description	Clear	Set	Default
I/O and Printing				
–33	I/O Device	Wire	IR	Wire
–34	Printing Device	IR	Wire	IR
–35	I/O Data Format	ASCII	Binary	ASCII
–36	RECV Overwrite	New variable	Overwrite	New
–37	Double-spaced Print	Single	Double	Single
–38	Linefeed	Inserts LF	Suppresses LF	Inserts
–39	Kermit Messages	Msg displayed	Msg suppressed	Clear
Time Management				
–40	Clock Display	TIME menu only	All times	Clear
–41	Clock Format	12 hour	24 hour	12 hour
–42	Date Format	MM/DD/YY	DD.MM.YY	Clear
–43	Rpt. Alarm Resched.	Rescheduled	Not rescheduled	Clear
–44	Acknowledged Alarms	Deleted	Saved	Deleted
Notes: If flag -43 is set, unacknowledged repeat alarms are not rescheduled. If flag -44 is set, acknowledged alarms are saved in the alarm catalog.				
Display Format				
–45→ –48	Set the number of digits in Fix, Scientific, and Engineering modes			0
Number Display Format		**–49**	**–50**	STD
–49 and –50	STD FIX SCI ENG	Clear Clear Set Set	Clear Set Clear Set	
–51	Fraction Mark	Decimal	Comma	Decimal
–52	Single Line Display	Multi-line	Single-line	Multi
–53	Precedence	() suppressed	() displayed	Clear
Miscellaneous				
–54	Tiny Array Elements	Replaces “tiny” pivots with 0	No replacement	Replaces
–55	Last Arguments	Saved	Not saved	Saved
–56	Beep	On	Off	On
–57	Alarm Beep	On	Off	On
–58	Verbose Messages	On	Off	On
–59	Fast Catalog Display	Off	On	Off
–60	Alpha Key Action	Twice to lock	Once to lock	Twice
–61	USR Key Action	Twice to lock	Once to lock	Twice
–62	User Mode	Not Active	Active	Clear
–63	Vectored Enter	Off	On	Off
–64	Set by GETI or PUTI when their element indices wrap around			
Equation Library				
60	Units Type	SI units	English units	SI
61	Units Usage	Units used	Units not used	Used
Multiple Equation Solver				
63	Variable State Change	↱ recalls variable	↱ toggles variable state	Recalls

Appendix D

Object Structures

This appendix describes the structure of some HP 48 objects. It is beyond the scope of this book to address the detailed structure of directories and libraries, so they are omitted here.

Sizes are expressed in nibbles. Prologues are always 5 nibbles, and unless otherwise noted field sizes (like a length or dimension count) are 5 nibbles. Length fields are self-relative lengths in nibbles. A length field for a 3 character string is 5 (length of length field) + 6 (number of nibbles in the string body) = 11.

D.1 Binary Integer

Atomic	Size = 10
Prologue	**Body**
`DOBINT`	5 nibbles

D.2 Real Number

Atomic			Size = 21
Prologue	**Exponent**	**Mantissa**	**Sign**
`DOREAL`	3 nibbles	12 nibbles	1 nibble

The exponent is encoded in tens complement form. A decimal point is implied between the first and second digits of the mantissa. The sign nibble is 0 for positive numbers or 9 for negative numbers.

D.3 Extended Real Number

Atomic			Size = 26
Prologue	**Exponent**	**Mantissa**	**Sign**
`DOREAL`	5 nibbles	15 nibbles	1 nibble

The exponent is encoded in tens complement form. A decimal point is implied between the first and second digits of the mantissa. The sign nibble is 0 for positive numbers or 9 for negative numbers.

D.4 Complex Number

Atomic		Size = 37
Prologue	**Real Part**	**Imaginary Part**
`DOCMP`	16 nibble real number body	16 nibble real number body

The real and imaginary parts are encoded using the format of the body of a real number object.

D.5 Extended Complex Number

Atomic		Size = 47
Prologue	**Real Part**	**Imaginary Part**
`DOCMP`	21 nibble real number body	21 nibble real number body

The real and imaginary parts are encoded using the format of the body of a real number object.

D.6 Character

Atomic	Size = 7
Prologue	**Body**
`DOCHAR`	2 nibbles

D.7 String

Atomic		Size = 10+2*number_of_characters
Prologue	**Length**	**Body**
`DOCSTR`	5 nibbles	Characters

D.8 Hex String

User binary integers (type 10) are implemented as hex strings.

Atomic		Size = 10+body_size
Prologue	**Length**	**Body**
`DOHSTR`	5 nibbles	Nibbles

D.9 Arrays

While array objects are structured to support an arbitrary number of dimensions, the kernel support is only meaningful for one or two dimension arrays. Arrays can be composed of most atomic object types.

D.9.1 One-Dimension Array

Atomic					Size = 25+$\sum$(object body sizes)
Prologue	**Length**	**Type Prologue**	**Dimension Count**	**Dimension Size**	**Object Bodies**
`DOARRY`	5 nibbles	5 nibbles	5 nibbles	5 nibbles	...

D.9.2 Two-Dimension Array

Atomic						Size = 30+$\sum$(object body sizes)
Prologue	**Length**	**Type Prologue**	**Dimension Count**	**1st Dimension Size**	**2nd Dimension Size**	**Object Bodies (row order)**
`DOARRY`	5 nibbles	5 nibbles	5 nibbles	5 nibbles	5 nibbles	...

D.9.3 Linked Array

A linked array is structured like the arrays above, but includes a table of pointers to object bodies. A one dimensional linked array looks like this:

Atomic						Size = 25+5*(number of objects)+∑(object body sizes)
Prologue	**Length**	**Type Prologue**	**Dimension Count**	**Dimension Size**	**Pointer Table**	**Object Bodies**
`DOARRY`	5 nibbles	5 nibbles	5 nibbles	5 nibbles	5*(#obs)	...

D.10 Name Objects

D.10.1 Global Name

Atomic		Size = 7+2*number_of_characters
Prologue	**Character Count**	**Body**
`DOIDNT`	2 nibbles	Characters

D.10.2 Local Name

Atomic		Size = 7+2*number_of_characters
Prologue	**Character Count**	**Body**
`DOLAM`	2 nibbles	Characters

D.10.3 XLIB Name

Atomic		Size = 11
Prologue	**Library Number**	**Object Number**
`DOROMP`	3 nibbles	3 nibbles

D.11 Graphic Object

Atomic				Size = 20+Height*CEIL(Width/8)
Prologue	**Length**	**Pixel Height**	**Pixel Width**	**Grob data in row order**
`DOGROB`	5 nibbles	5 nibbles	5 nibbles	...

Graphic objects store data in row order, and the rows must have even byte widths. The bits in each nibble are reversed — the most significant bit represents the rightmost pixel.

D.12 Code Object

Atomic		Size = 10+body_size
Prologue	**Length**	**Body**
`DOCODE`	5 nibbles	Nibbles

D.13 Secondary

Composite		Size = 10+∑(object sizes)
Prologue	**Body**	**SEMI**
`DOCOL`	... objects ...	5 nibbles

D.14 Tagged

Atomic	Size = 12+2*number_of_characters+object_size			
Prologue	**Tag Length**	**Tag**	**Object**	**SEMI**
`DOTAG`	2 nibbles	Characters	...	5 nibbles

NOTE: A tagged object *is* considered atomic, and cannot be decomposed with `INNERCOMP`.

D.15 List

Composite	Size = 10+∑(object sizes)	
Prologue	**Body**	**SEMI**
`DOLIST`	... objects ...	5 nibbles

D.16 Symbolic

Composite	Size = 10+∑(object sizes)	
Prologue	**Body**	**SEMI**
`DOSYMB`	... objects ...	5 nibbles

D.17 Unit

Composite	Size = 31+∑(object sizes)		
Prologue	**Real Number**	**Body**	**umEND**
`DOEXT`	21 nibbles	... objects ...	5 nibbles

D.18 Library Data Objects

A Library Data object is a “generic bucket” into which an arbitrary set of data may be stored. This object type is used by Equation Library applications, like the Multiple Equation Solver, the MineHunt game, and the Periodic Table application.

Atomic	Size = 10+body_size	
Prologue	**Length**	**Body**
`DOEXT0`	5 nibbles	Nibbles

To avoid conflicts between applications, HP uses a convention for storing a list of information into a library data object. The information stored is actually a list consisting of a bint and another object, typically a list. The first five nibbles of the body encode the ROMID of the parent application. To illustrate this, consider Mpar, a library data object used by the Multiple Equation Solver. Mpar looks like this:

Prologue	**Length**	**RomId**	**Rest of Body**			
`DOEXT0`	5 nibbles	5 nibbles	`DOLIST`	Mpar Objects	SEMI	SEMI

When Mpar is recalled by the Multiple Equation solver, it is copied to TEMPOB. If the ROMID matches the ROMID of the Multiple Equation Solver the first part of the object is overwritten with the prologue for a list and bint as follows:

`DOLIST`	`DOBINT`	RomId (5 nibbles)	`DOLIST`	Mpar Objects	SEMI	SEMI

The object `MESRclEqn` does this job for the Multiple Equation Solver:

`MESRclEqn`	#E4012h	G/GX XLIB 228 18
Recalls the contents of the reserved variable Mpar		
	→	{ equation list }

Index

::N, 100
::NEVAL, 100
<DelKey, 141
<SkipKey, 140
>DelKey, 141
>HCOMP, 100
>H$, 94
>R, 51
>SkipKey, 141
>TAG, 91
>TCOMP, 100
>T$, 94
?CaseKeyDef, 155
?DispMenu, 115
?DispStack, 114
?DispStatus, 114
?PURGE_HERE, 108
?SEMI, 38
?SKIP, 39
?SKIPSWAP, 40
#*, 26
#+, 26
#+-1, 26
#+DUP, 27
#+OVER, 27
#+PICK, 68
#+ROLL, 27, 67
#+SWAP, 27
#+UNROLL, 67
#-, 26
#-+1, 27
#-ROLL, 67
#-SWAP, 27
#-UNROLL, 27, 67
#-#2/, 27
#/, 26
#0<>, 33
#0=, 33
#0=?SEMI, 38
#0=?SEKIP, 40
#0=ITE, 40
#1+, 26
#1+DUP, 27
#1+LAST$, 93
#1+NDROP, 27, 67
#1+PICK, 27, 67
#1+ROLL, 27, 68
#1+ROT, 28
#1+SWAP, 28
#1+UNROLL, 28, 68
#1+_ONE_DO, 48
#1-, 26
#1-1SWAP, 28
#1-DUP, 28
#1-ROT, 28
#1-SUB$, 93
#1-SWAP, 28
#1-UNROT, 28
#10*, 26
#10+, 26
#12+, 26
#1<>, 33
#1=, 33
#1=?SKIP, 40
#2*, 26
#2+, 26
#2+PICK, 28, 68
#2+ROLL, 28, 68
#2+UNROLL, 28, 68
#2-, 26
#2/, 26
#2<>, 33
#2=, 33
#3+, 26
#3+PICK, 28, 68
#3-, 26
#3=, 33
#4+, 26
#4+PICK, 28, 68
#4-, 26
#5+, 26
#5-, 26
#5=, 33
#8*, 26
#8+, 26
#:>$, 93
#<, 32
#<>, 32
#<ITE, 40
#=, 32
#=ITE, 40
#=casedrop, 43, 44, 153
#=casedropfls, 154
#>, 32
#>?SKIP, 40
#>CHR, 91
#>HXS, 96
#>$, 93
#FIVE#FOUR, 25
#NAME?, 48
#ONE#2, 25
#THREE#FOUR, 25
#TWO#FOUR, 25
#TWO#ONE, 25
#TWO#TWO, 25
#ZERO#ONE, 25
#ZERO#SEVEN, 25
$>BIGGROB, 125
$>GROB, 125
$>ID, 93, 108
$>grob, 125
$_2DQ, 93
$_::, 93
$_<<>>, 93
$_ECHO, 93
$_EXIT, 93
$_GRAD, 93
$_LRParens, 93
$_R<Z, 93
$_R<<, 93
$_RAD, 93
$_Undefined, 93
$_XYZ, 93
$_[], 93
$_'', 93
$_{}, 93
%, 80
%-1, 79
%-2, 79
%-3, 79
%-4, 79
%-5, 79
%-6, 79
%-7, 79
%-8, 79
%-9, 79
%-MAXREAL, 79
%-MINREAL, 79
%.1, 79
%.5, 79
%0, 79
%0<, 34
%0<>, 31, 34
%0=, 34
%0=case, 44
%0>, 34
%0>=, 34
%1, 79
%1+, 80
%1-, 80
%1/, 80
%10, 79
%10*, 80
%100, 79
%11, 79
%12, 79
%13, 79
%14, 79
%15, 79
%16, 79
%17, 79
%18, 79
%180, 79
%19, 79
%1=case, 44

%2, 79
%20, 79
%21, 79
%22, 79
%23, 79
%24, 79
%25, 79
%26, 79
%27, 79
%3, 79
%360, 79
%4, 79
%5, 79
%6, 79
%7, 79
%8, 79
%9, 79
%<, 33
%<=, 33
%<>, 34
%=, 34
%>, 34
%>=, 34
%>C%, 86
%>HMS, 81
%>TAG, 90
%>#, 96
%>%%, 80
%>%%1/, 84
%ABS, 80
%ASIN, 80
%ASINH, 80
%ATAN, 80
%ATANH, 81
%CEIL, 81
%CH, 81
%CHS, 81
%COMB, 81
%COS, 81
%COSH, 81
%D>R, 81
%EXP, 81
%EXPM1, 81
%EXPONENT, 81
%FACT, 81
%FLOOR, 81
%FP, 81
%HMS+, 81
%HMS-, 81
%HMS>, 81
%IP, 82
%LN, 82
%LNP1, 82
%LOG, 82
%MANTISSA, 82
%MAX, 82
%MAXREAL, 79
%MAXorder, 34
%MIN, 82
%MINREAL, 79
%MOD, 82
%NFACT, 82
%NROOT, 82
%OF, 82
%PERM, 82
%PI, 79
%POL>%REC, 82
%R>D, 82
%RAN, 82
%RANDOMIZE, 82
%REC>%POL, 82
%SGN, 83
%SIN, 83
%SINH, 83
%SPH>%REC, 83
%SQRT, 83
%T, 83
%TAN, 83
%TANH, 83
%%, 83
%%*, 83
%%*ROT, 83
%%*SWAP, 83
%%+, 84
%%-, 84
%%.1, 79
%%.4, 79
%%.5, 79
%%/, 84
%%/>%, 84
%%0, 79
%%0<, 35
%%0<=, 35
%%0<>, 35
%%0=, 35
%%0>, 35
%%0>=, 35
%%1, 79
%%1/, 84
%%10, 79
%%12, 79
%%2, 79
%%2PI, 79
%%3, 79
%%4, 79
%%5, 79
%%60, 79
%%7, 79
%%<, 34
%%<=, 34
%%>, 34
%%>=, 34
%%>C%, 86
%%>%, 80
%%ABS, 84
%%ACOSRAD, 84
%%ANGLE, 84
%%ANGLEDEG, 84
%%ANGLERAD, 84
%%ASINRAD, 84
%%CHS, 84
%%COS, 84
%%COSDEG, 84
%%COSH, 84
%%COSRAD, 84
%%EXP, 85
%%FLOOR, 85
%%H>HMS, 85
%%INT, 85
%%LN, 85
%%LNP1, 85
%%MAX, 85
%%P>R, 85
%%PI, 79
%%R>P, 85
%%SIN, 85
%%SINDEG, 85
%%SINH, 85
%%SQRT, 85
%%TANRAD, 85
%%^, 84
%^, 83
%e, 79
&COMP, 101
&HXS, 97
&$, 93
&$SWAP, 93
', 51
'DoBadKey, 156
'DoBadKeyT, 156
'R, 51
append$, 93
append$SWAP, 93
{}N, 99
0LASTOWDOB, 52
10GETLAM, 61
10PUTLAM, 61
10UNROLL, 68
11GETLAM, 61
11PUTLAM, 61
12GETLAM, 61
12PUTLAM, 61
13GETLAM, 61
13PUTLAM, 61
14GETLAM, 61
14PUTLAM, 61
15GETLAM, 61
15PUTLAM, 61
16GETLAM, 61
16PUTLAM, 61
17GETLAM, 61

17PUTLAM, 61
18GETLAM, 61
18PUTLAM, 61
19GETLAM, 61
19PUTLAM, 61
1ABNDSWAP, 63
1GETABND, 63
1GETLAM, 61
1GETSWAP, 63
1LAMBIND, 64
1NULLLAM{}, 64
1PUTLAM, 61
1_#1-SUB$, 93
20GETLAM, 61
20PUTLAM, 61
21GETLAM, 61
21PUTLAM, 61
22GETLAM, 61
22PUTLAM, 61
2DROP, 68
2DROP00, 27, 68
2DROPFALSE, 68
2DUP, 68
2DUP5ROLL, 68
2DUPEQ, 32
2DUPSWAP, 68
2DUP#+, 27
2DUP#<, 32
2DUP#=, 32
2DUP#>, 33
2EXT, 57
2GETEVAL, 64
2GETLAM, 61
2GROB, 57
2HXSLIST?, 96, 121
2LIST, 57
2OVER, 68
2Ob>Seco, 100
2PUTLAM, 61
2RDROP, 52
2REAL, 57
2SWAP, 68
2%>%%, 80
2%%>%, 80
3DROP, 68
3GETLAM, 61
3PICK, 68
3PICK3PICK, 68
3PICKOVER, 68
3PICKSWAP, 68
3PICK#+, 27
3PUTLAM, 61
3RDROP, 52
3REAL, 57
3UNROLL, 68
4DROP, 68
4GETLAM, 61
4NULLLAM{}, 64
4PICK, 68
4PICKOVER, 68
4PICKSWAP, 68
4PICK#+, 27
4PICK#+SWAP, 27
4PUTLAM, 61
4ROLL, 69
4ROLLDROP, 69
4ROLLOVER, 69
4ROLLROT, 69
4ROLLSWAP, 69
4UNROLL, 69
4UNROLL3DROP, 69
4UNROLLDUP, 69
4UNROLLROT, 69
5DROP, 69
5GETLAM, 61
5PICK, 69
5PUTLAM, 61
5ROLL, 69
5ROLLDROP, 69
5UNROLL, 69
6DROP, 69
6GETLAM, 61
6PICK, 69
6PUTLAM, 61
6ROLL, 69
6UNROLL, 69
7DROP, 69
7GETLAM, 61
7PICK, 69
7PUTLAM, 61
7ROLL, 69
7UNROLL, 69
8GETLAM, 61
8PICK, 69
8PUTLAM, 61
8ROLL, 69
8UNROLL, 69
9GETLAM, 61
9PUTLAM, 61

a%>$, 96
a%>$,, 96
ABND, 59
ABORT, 66
ABUFF, 116
AGAIN, 50
AllowIntr, 233
AND, 31
AND$, 94
ANDcase, 44
ANDITE, 40
ANDNOTcase, 44
any, 56
apndvarlst, 100
arry, 56
ARSIZE, 89
ATTN?, 134
ATTNFLGCLR, 134
AtUserStack, 53

BEGIN, 50
BigCursor, 129
BIGDISPN, 124
BIGDISPROW1, 124
BIGDISPROW2, 124
BIGDISPROW3, 124
BIGDISPROW4, 124
BIND, 58
BINT255d, 25
BINT_131d, 25
bit*, 98
bit+, 98
bit-, 98
bit/, 98
bit#%*, 98
bit#%+, 98
bit#%-, 98
bit#%/, 98
bit%#*, 98
bit%#+, 98
bit%#-, 98
bit%#/, 98
bitAND, 98
bitASR, 98
bitNOT, 98
bitOR, 98
bitRL, 98
bitRLB, 99
bitRR, 99
bitRRB, 99
bitSL, 99
bitSLB, 99
bitSR, 99
bitSRB, 99
bitXOR, 99
Blank$, 94
BlankDA1, 116
BlankDA12, 116
BlankDA2, 116
BLANKIT, 115
Box/StdLabel, 128
BRbrowse, 170
BRDispItems, 171
BRdone, 170
BRGetItem, 171
BRinverse, 171
BRoutput, 171
BRRclC1, 171
BRRclCurRow, 171
BRStoC1, 171
BRViewItem, 171

C>Im%, 86

C>Re%, 86
C%0=, 35
C%1/, 87
C%>#, 121
C%>%, 86
C%>%%, 86
C%>%%SWAP, 86
C%%0=, 35
C%%>%%, 86
C%%>C%, 86
C%%CHS, 87
C%%CONJ, 88
C%ABS, 87
C%ACOS, 87
C%ACOSH, 87
C%ALOG, 87
C%ARG, 87
C%ASIN, 87
C%ASINH, 87
C%ATAN, 87
C%ATANH, 87
C%C^C, 87
C%C^R, 87
C%CHS, 87
C%CONJ, 88
C%COS, 88
C%COSH, 88
C%EXP, 88
C%LN, 88
C%LOG, 88
C%R^C, 88
C%SGN, 88
C%SIN, 88
C%SINH, 88
C%SQRT, 88
C%TAN, 88
C%TANH, 88
CACHE, 64
CAR$, 94
CARCOMP, 101
case, 41
case2DROP, 47
case2drpfls, 47
caseDoBadKey, 47
caseDROP, 43, 47
casedrop, 42, 46
caseDrpBadKy, 47
casedrpfls, 47
casedrptru, 48
caseERRJMP, 47
caseFALSE, 47
caseSIZEERR, 47
caseTRUE, 47
CDR$, 94
CDRCOMP, 101
char, 56
CHECKHEIGHT, 117
CHECKKEY, 133
CHECKPVARS, 122
ChkLowBat, 228
Choose, 174
ChooseMenu0, 179
ChooseMenu1, 179
ChooseMenu2, 179
CHR>$, 91
CHR>#, 91
CHR>$, 94
CHR_*, 92
CHR_+, 92
CHR_,, 92
CHR_-, 92
CHR_->, 92
CHR_., 92
CHR_..., 92
CHR_/, 92
CHR_0, 92
CHR_00, 92
CHR_1, 92
CHR_2, 92
CHR_3, 92
CHR_4, 92
CHR_5, 92
CHR_6, 92
CHR_7, 92
CHR_8, 92
CHR_9, 92
CHR_:, 92
CHR_;, 92
CHR_<, 92
CHR_«, 92
CHR_<=, 92
CHR_<>, 92
CHR_=, 92
CHR_>, 92
CHR_>=, 92
CHR_», 92
CHR_[, 92
CHR_#, 92
CHR_{, 92
CHR_}, 92
CHR_], 92
CHR_A, 92
CHR_a, 92
CHR_Angle, 92
CHR_B, 92
CHR_b, 92
CHR_C, 92
CHR_c, 92
CHR_D, 92
CHR_d, 92
CHR_DblQuote, 92
CHR_Deriv, 92
CHR_E, 92
CHR_e, 92
CHR_F, 92
CHR_f, 92
CHR_G, 92
CHR_g, 92
CHR_H, 92
CHR_h, 92
CHR_I, 92
CHR_i, 92
CHR_Integral, 92
CHR_J, 92
CHR_j, 92
CHR_K, 92
CHR_k, 92
CHR_L, 92
CHR_l, 92
CHR_LeftPar, 92
CHR_M, 92
CHR_m, 92
CHR_N, 92
CHR_n, 92
CHR_Newline, 92
CHR_O, 92
CHR_o, 92
CHR_P, 92
CHR_p, 92
CHR_Pi, 92
CHR_Q, 92
CHR_q, 92
CHR_R, 92
CHR_r, 92
CHR_RightPar, 92
CHR_S, 92
CHR_s, 92
CHR_Sigma, 92
CHR_Space, 92
CHR_T, 92
CHR_t, 92
CHR_U, 92
CHR_u, 92
CHR_UndScore, 92
CHR_V, 92
CHR_v, 92
CHR_W, 92
CHR_w, 92
CHR_X, 92
CHR_x, 92
CHR_Y, 92
CHR_y, 92
CHR_Z, 92
CHR_z, 92
CK0, 53
CK0NOLASTWD, 53
CK1, 53
CK1&Dispatch, 55
CK1NOLASTWD, 53
CK2, 53
CK2&Dispatch, 55

CK2NOLASTWD, 53
CK3, 53
CK3&Dispatch, 55
CK3NOLASTWD, 53
CK4, 53
CK4&Dispatch, 55
CK4NOLASTWD, 53
CK5, 53
CK5&Dispatch, 55
CK5NOLASTWD, 53
Ck&DecKeyLoc, 138
CK&DISPATCH0, 54
CK&DISPATCH1, 54
CKGROBFITS, 126
CKN, 53
CKNNOLASTWD, 53
CKREF, 102, 111
CLEARMENU, 119
CLEARVDISP, 116
Clr16, 116
Clr8, 116
Clr8-15, 116
ClrDA1IsStat, 113, 116
ClrDAsOK, 115
cmp, 56
CodePl>%rc.p, 138
COERCE, 25
COERCE2, 26
COERCE$22, 94
COERCEFLAG, 30
COLA, 51
CONTEXT, 110
CREATE, 108
CREATEDIR, 110
CROSSGROB, 129
CURSOR1, 129
CURSOR2, 129

Date>d$, 94
DDAYS, 83
Debounce, 235
DECOMP$, 94
DEPTH, 69
DisableIntr, 233
Disp5x7, 123
DispEditLine, 114
DispMenu, 114, 119, 164
DispMenu.1, 115, 119, 164
DISPN, 123
DISPROW1, 123
DISPROW1*, 123
DISPROW2, 123
DISPROW2*, 123
DISPROW3, 123
DISPROW4, 123
DISPROW5, 123
DISPROW6, 123
DISPROW7, 123
DispStatus, 114
DISPSTATUS2, 123
DO, 48
DO$EXIT, 67
DO>LCD, 116
DO#EXIT, 67
DoBadKey, 156
DOBIND, 59
DOC>PX, 121
DoCKeyCancel, 180
DoCKeyChAll, 180
DoCKeyCheck, 180
DoCKeyOK, 180
DoCKeyUnChAll, 180
DOCLLCD, 116
DoInputForm, 184
DoKeyCancel, 191
DoKeyOK, 191
DOLCD>, 116
DoMsgBox, 168
DoNewMatrix, 90
DoOldMatrix, 90
DOPX>C, 121
DOVARS, 110
DROP, 69
DROP#1-, 28
DROPDUP, 70
DROPFALSE, 30, 70
DROPNDROP, 70
DROPNULL$, 70
DROPNULL$, 94
DROPONE, 28, 70
DROPOVER, 70
DROPROT, 70
DROPSWAP, 70
DROPSWAPDROP, 70
DROPTRUE, 30, 70
DROPZERO, 70
DUMP, 64
DUP, 70
DUP1LAMBIND, 64
DUP3PICK, 70
DUP3PICK#+, 28
DUP4PUTLAM, 64
DUP4UNROLL, 70
DUP#0<>, 33
DUP#0=, 33
DUP#0=case, 44
DUP#0=csDROP, 46
DUP#0=csedrp, 42, 45
DUP#0=IT, 40
DUP#0=ITE, 40
DUP#0_DO, 48
DUP#1+, 28
DUP#1+PICK, 28, 70
DUP#1-, 29
DUP#1=, 33
DUP#7<, 33
DUP$>ID, 94, 108
DUP%0=, 34
DUPDUP, 70
DUPGROBDIM, 125
DUPINCOMP, 101
DUPLEN$, 94
DUPLENCOMP, 100
DUPNULL$?, 37
DUPNULL$?, 94
DUPONE, 70
DUPPICK, 70
DUPROLL, 70
DUPROT, 70
DUPTEMPENV, 64
DUPTWO, 29, 70
DUPTYPEARRY?, 57
DUPTYPEBINT?, 57
DUPTYPECHAR?, 57
DUPTYPECMP?, 57
DUPTYPECOL?, 57
DUPTYPECSTR?, 57
DUPTYPEEXT?, 57
DUPTYPEGROB?, 57
DUPTYPEHSTR?, 57
DUPTYPEIDNT?, 57
DUPTYPELAM?, 57
DUPTYPELIST?, 57
DUPTYPEREAL?, 57
DUPTYPEROMP?, 57
DUPTYPERRP?, 57
DUPTYPESYMB?, 57
DUPTYPETAG?, 57
DUPUNROT, 70
DUPZERO, 29, 70

ECUSER, 226
EDITDECOMP$, 94
EditMenu, 140
EIGHT, 25
EIGHTEEN, 25
EIGHTY, 25
EIGHTYONE, 25
ELEVEN, 25
Embedded?, 102
EQ, 32
EQcase, 45
EQIT, 40
EQITE, 41
EQOR, 32
EQOVER, 32
EQUAL, 32
EQUALcase, 45
EQUALcasedrp, 45
EQUALNOT, 32
EQUALNOTcase, 45
EQUALOR, 32
EQUALPOSCOMP, 102

ERRBEEP, 67
ERRJMP, 65
Errjmp, 228
ERRORCLR, 66
ERROROUT, 67
ERRORSTO, 66
ERRSET, 65
ERRTRAP, 65
EXITMSGSTO, 66
EXTN, 100, 104
EXTOBOB, 57
EXTREAL, 57
EXTSYM, 57

FALSE, 30
FalseFalse, 30
FALSETRUE, 30
FIFTEEN, 25
FIFTY, 25
FIFTYEIGHT, 25
FIFTYFIVE, 25
FIFTYFOUR, 25
FIFTYNINE, 25
FIFTYONE, 25
FIFTYSEVEN, 25
FIFTYSIX, 25
FIFTYTHREE, 25
FIFTYTWO, 25
FIVE, 25
FlashMsg, 123
FlashWarning, 123
FLUSHKEYS, 133
FORTY, 25
FORTYEIGHT, 25
FORTYFIVE, 25
FORTYFOUR, 25
FORTYNINE, 25
FORTYONE, 25
FORTYSEVEN, 25
FORTYSIX, 25
FORTYTHREE, 25
FORTYTWO, 25
FOUR, 25
FOURTEEN, 25

GARBAGE, 111
GBUFF, 117
GBUFFGROBDIM, 117, 125
GDISP, 117
GETATELN, 89
GETEXITMSG, 66
GetFieldVals, 191
GETLAM, 61
GETLAMPAIR, 64
getnibs, 112
GETPMIN&MAX, 122
GETPTR, 217
GETPTRLOOP, 217
GETSCALE, 122
GETTEMP, 224
GETTOUCH, 133
GETXMAX, 122
GETXMIN, 122
GETYMAX, 122
GETYMIN, 122
gFldVal, 191
GPErrjmpC, 228
GPOverWrALp, 223
GPOverWrFLp, 222
GPOverWrR0Lp, 223
GPOverWrT/FL, 222
GPOverWrTLp, 221
GPPushFLoop, 222
GPPushT/FLp, 222
GPPushTLoop, 222
grob, 56
GROB+#, 126
Grob>Menu, 128
GROB, 126
GROBZERO, 126
GROBZERODRP, 126
grobAlertIcon, 168
grobCheckKey, 180
GROBDIM, 125
GROBDIMw, 125

HARDBUFF, 115
HARDBUFF2, 119
HEIGHTENGROB, 117
hxs, 56
HXS<=HXS, 37, 97
HXS<HXS, 37, 97
HXS==HXS, 37, 97
HXS>=HXS, 37, 97
HXS>#, 96
HXS>$, 97
hxs>$, 97
HXS>%, 96
HXS>HXS, 37, 97
HXS#HXS, 37, 97

ID>$, 95, 108
Id>Menu, 128
ID>TAG, 91
idnt, 56
IDREAL, 57
IDUP, 51
IFMenuRow1, 192
IFMenuRow2, 192
INCOMPDROP, 101
INDEXSTO, 49
InitEnab, 226
InitMenu, 164
InitMenu%, 164
INNERCOMP, 101
INNERDUP, 101
InputLine, 140
INTEMNOTREF?, 111
INVGROB, 126
IStackKey, 141
ISTOPSTO, 49
IT, 39
ITE, 39

JINDEXSTO, 49
JstGETTHEMESG, 94
JstGETTHEMSG, 67
JSTOPSTO, 49
JUMPBOT, 118
JUMPLEFT, 118
JUMPRIGHT, 118
JUMPTOP, 118

KEYINBUFFER?, 133

lam, 56
LAST$, 95
LEDispItem, 182
LEDispList, 182
LEDispPrompt, 182
LEN$, 95
LENCOMP, 100
LENHXS, 97
LINECHANGE, 119
LINEOFF, 127
LINEOFF3, 127
LINEON, 127
LINEON3, 127
list, 56
LISTCMP, 57
LISTLAM, 57
LISTREAL, 57
LOOP, 48

MABS, 104
MAKE$, 224
MAKE$N, 224
MAKEARRY, 89
makebeep, 229
MakeBoxLabel, 128
MakeDirLabel, 128
MAKEGROB, 125
MakeInvLabel, 128
MAKEPICT#, 117
MakeStdLabel, 128
MARKGROB, 129
matchob?, 102
MATCON, 89
MATREDIM, 89
MATTRN, 89
MDIMS, 90
MDIMSDROP, 90
MediumCursor, 129
MEM, 112

MENUOFF?, 119
MESRclEqn, 258
MINUSONE, 25
MOVEDOWN, 226
MOVEUP, 226
MsgBoxMenu, 168

N+1DROP, 70
NcaseSIZEERR, 46
NcaseTYPEERR, 46
NDROP, 70
NDROPFALSE, 70
NDUP, 70
NDUPN, 71
NEWLINE$, 93
NEWLINE$&$, 95
NINE, 25
NINETEEN, 25
NoExitAction, 180
NOT, 31
NOT?SEMI, 38
NOT_IT, 39
NOTAND, 31
NOTcase, 45
NOTcase2DROP, 46
NOTcase2drop, 45
NOTcasedrop, 45
NOTcaseFALSE, 46
NOTcaseTRUE, 46
NTHCOMDDUP, 101
NTHCOMPDROP, 101
NTHELCOMP, 101
NTHOF, 102
NULL$?, 37
NULL::, 99
NULL$, 95
NULL$?, 95
NULL$TEMP, 95
NULL{}, 99
NULLHXS, 97
NULLLAM, 61
NullMenuKey, 141, 158
NULL$SWAP, 95

Ob>Seco, 100
OCRC, 112
OCRC%, 112
ONE, 25
ONE{}N, 99
ONEFALSE, 71
ONEHUNDRED, 25
ONEONE, 25
ONESWAP, 71
OR$, 95
ORcase, 46
ORDERXY#, 127
ORNOT, 31
OVER, 71
OVER5PICK, 71
OVER#+, 29
OVER#-, 29
OVER#0=, 33
OVER#2+UNROL, 29
OVER#=case, 46
OVERARSIZE, 90
OVERDUP, 71
OVERLEN$, 95
OVERSWAP, 71
OVERUNROT, 71
OverWrF/TLp, 222
OverWrFLoop, 222
OverWrT/FLp, 222
OverWrTLoop, 222

palparse, 96
ParOuterLoop, 145
PATHDIR, 110
PICK, 71
PIXOFF, 127
PIXOFF3, 127
PIXON, 127
PIXON3, 127
PIXON?, 127
PIXON?3, 127
POLKeyUI, 163
POLRestoreUI, 163
POLResUI&Err, 163
POLSaveUI, 163
POLSetUI, 163
POP1%, 220
POP2#, 220
POP2%, 220
POP#, 220
PopASavptr, 220
popflag, 220
POS$, 95
POS$REV, 95
POSCOMP, 102
PromptIdUtil, 95
PuHiddenVar, 110
PULLCMPEL, 90
PULLREALEL, 90
PURGE, 108
PUSH2#, 221
PUSH#, 221
PUSH#ALOOP, 221
PUSH#LOOP, 221
PUSH%, 221
LOOPPUSH%LOOP, 221
PUSHA, 221
PushF/TLoop, 223
PushFLoop, 222
PushT/FLoop, 223
PushTLoop, 222
PUTCMPEL, 90
PUTEL, 90
PUTLAM, 61
PUTLIST, 101
putnibs, 112
PUTREALEL, 90
PUTSCALE, 122
PUTXMAX, 122
PUTXMIN, 122
PUTYMAX, 122
PUTYMIN, 122

R>, 52
RclHiddenVar, 110
RDROP, 52
RDUP, 52
Re>C%, 86
real, 56
REALEXT, 57
REALOB, 57
REALOBOB, 57
REALREAL, 57
REALSYM, 57
RECLAIMDISP, 116
REPEAT, 50
REPEATER, 138
REPEATERCH, 138
reversym, 72
RNDXY, 83
ROLL, 71
ROLLDROP, 71
ROLLSWAP, 71
ROMPANY, 57
ROT, 71
ROT2DROP, 71
ROT2DUP, 71
ROT#+, 29
ROT#+SWAP, 29
ROT#-, 29
ROT#1+, 29
ROTAND, 31
ROTDROP, 71
ROTDROPSWAP, 71
ROTDUP, 71
ROTOVER, 71
ROTROT2DROP, 71
ROTSWAP, 71
RPIT, 39
RPITE, 39
RSWAP, 52

SAFESTO, 109
SAVPTR, 217
SCROLLDOWN, 118
SCROLLLEFT, 118
SCROLLRIGHT, 118
SCROLLUP, 118
seco, 56
Seco>Menu, 128
SEP$NL, 95

SetDA12Temp, 115
SetDA1Bad, 114
SetDA1Temp, 114
SetDA2aBad, 114
SetDA2aTemp, 114
SetDA2bBad, 114
SetDA2bTemp, 114
SetDA2OKTemp, 114
SetDA3Bad, 115
SetDA3Temp, 115
SetDAsTemp, 115
SETIVLERR, 67
SETMEMERR, 67
SETNONEXTERR, 67
SETSIZEERR, 67
SETSTACKERR, 67
SETTYPEERR, 67
SEVEN, 25
SEVENTEEN, 25
SEVENTY, 25
SEVENTYFOUR, 25
SEVENTYNINE, 25
SIX, 25
SIXTEEN, 25
SIXTY, 25
SIXTYEIGHT, 25
SIXTYFOUR, 25
SIXTYONE, 25
SIXTYTHREE, 25
SIXTYTWO, 25
SmallCursor, 129
SPACE$, 93
Std/BoxLabel, 128
STO, 59, 109
StoHiddenVar, 110
str, 56
Str>Menu, 128
STRIPTAGS, 91
STRIPTAGSl2, 91
SUB$, 95
SUB$1#, 95
SUB$SWAP, 95
SUBCOMP, 101
SUBGROB, 126
SUBHXS, 97
SWAP, 71
SWAP2DUP, 71
SWAP3PICK, 71
SWAP4PICK, 71
SWAP4ROLL, 71
SWAP#-, 29
SWAP#1+, 29
SWAP#1+SWAP, 29
SWAP#1-, 29
SWAP#1-SWAP, 29
SWAP%>C%, 86
SWAP&$, 96
SWAPCKREF, 111
SWAPDROP, 71
SWAPDROPDUP, 72
SWAPDROPSWAP, 72
SWAPDROPTRUE, 72
SWAPDUP, 72
SWAPINCOMP, 101
SWAPONE, 72
SWAPOVER, 72
SWAPOVER#-, 29
SWAPROT, 72
SWAPTRUE, 72
sym, 56
symb, 56
Symb>HBuff, 125
SYMBN, 100
SYMBUNIT, 57
SYMEXT, 57
SYMID, 57
SYMLAM, 57
SYMOB, 57
SYMREAL, 57
SYMSYM, 57
SYSCONTEXT, 110
SysDisplay, 113
SysITE, 41
SysSTO, 109

TAGGED, 56
TAGGEDANY, 57
TAGOBS, 91
TakeOver, 155
TEN, 25
THIRTEEN, 25
THIRTY, 25
THIRTYEIGHT, 25
THIRTYFIVE, 25
THIRTYFOUR, 25
THIRTYNINE, 25
THIRTYONE, 25
THIRTYSEVEN, 25
THIRTYSIX, 25
THIRTYTHREE, 25
THIRTYTWO, 25
THREE, 25
THREE{}N, 99
ticR, 51
TIMESTR, 96
TOADISP, 116
TOD>t$, 96
TOGDISP, 117
TogInsertKey, 141
TOGLINE, 127
TOGLINE3, 127
TOTEMPOB, 111
TRCXY, 83
TRUE, 30
TrueFalse, 30
TrueTrue, 30
TURNMENUOFF, 119
TURNMENUON, 119
TWELVE, 25
TWENTY, 25
TWENTYEIGHT, 25
TWENTYFIVE, 25
TWENTYFOUR, 25
TWENTYNINE, 25
TWENTYONE, 25
TWENTYSEVEN, 25
TWENTYSIX, 25
TWENTYTHREE, 25
TWENTYTWO, 25
TWO, 25
TWO{}N, 99
TYPEARRY?, 57
TYPEBINT?, 57
TYPECARRY?, 57
TYPECHAR?, 57
TYPECMP?, 57
TYPECOL?, 57
TYPECSTR?, 57
TYPEEXT?, 57
TYPEGROB?, 57
TYPEHSTR?, 57
TYPEIDNT?, 57
TYPELAM?, 57
TYPELIST?, 57
TYPERARRY?, 57
TYPEREAL?, 57
TYPEROMP?, 57
TYPERRP?, 57
TYPESYMB?, 57
TYPETAGGED?, 57

UM*, 104
um*, 103
UM+, 104
UM-, 104
UM/, 104
um/, 103
UM<=?, 36
UM<?, 36
UM=?, 36
UM>=?, 36
UM>?, 36
UM>U, 104
UM#?, 36
UM%, 104
UM%CH, 104
UM%T, 104
um^, 103
UMCEIL, 104
UMCHS, 104
UMCONV, 104
UMCOS, 104
umEND, 103

UMFLOOR, 104
UMFP, 104
UMIP, 104
UMMAX, 104
UMMIN, 105
umP, 103
UMRND, 105
UMSI, 105
UMSIGN, 105
UMSIN, 105
UMSQ, 105
UMSQRT, 105
UMTAN, 105
UMTRC, 105
UMU>, 105
UMXROOT, 105
UNCOERCE, 26
UNCOERCE2, 26
UNIT>$, 105
unitob, 56
UNROLL, 72
UNROT, 72
UNROT2DROP, 72
UNROTDROP, 72
UNROTDUP, 72
UNROTOVER, 72
UNROTSWAP, 72
UNROTSWAPDRO, 72
UNTIL, 50
UPDIR, 110
USER$>TAG, 91
UserITE, 41

WaitForKey, 137
WHILE, 50
WIDENGROB, 117
WINDOW#, 117
WINDOWCORNER, 118
WINDOWDOWN, 118
WINDOWLEFT, 118
WINDOWRIGHT, 118
WINDOWUP, 118
WINDOWXY, 118
WORDSIZE, 97

XEQORDER, 110
XEQPGDIR, 110
XEQSTOID, 109
XOR, 31
XOR$, 96
XYGROBDISP, 126

ZERO, 25
ZERO_DO, 49
ZEROISTOPSTO, 49
ZEROOVER, 29, 72
ZEROSWAP, 30, 72
ZEROZERO, 25
ZEROZEROONE, 25
ZEROZEROTWO, 25
ZEROZEROZERO, 25