CONTENTS

1. Introduction.. 1

2. RPL Principles...................................... 2
   2.1 Origins... 2
   2.2 Mathematical Control............................ 3
   2.3 Formal Definitions 5
   2.4 Execution....................................... 6
      2.4.1 EVAL...................................... 8
      2.4.2 Data Class Objects........................ 9
      2.4.3 Identifier Class Objects.................. 9
      2.4.4 Procedure Class Objects................... 10
      2.4.5 Object Skipover and SEMI.................. 10
      2.4.6 RPL Pointers.............................. 11
   2.5 Memory Management............................... 11
   2.6 User RPL and System RPL......................... 13
   2.7 Programming in System RPL....................... 14
   2.8 Sample RPL Program.............................. 16
      2.8.1 The Source File........................... 16
      2.8.2 Compiling the Program..................... 18

3. Object Structures................................... 19
   3.1 Object Types.................................... 19
      3.1.1 Identifier Object......................... 19
      3.1.2 Temporary Identifier Object............... 19
      3.1.3 ROM Pointer Object........................ 20
      3.1.4 Binary Integer Object..................... 20
      3.1.5 Real Number Object........................ 20
      3.1.6 Extended Real Number Object............... 21
      3.1.7 Complex Number Object..................... 22
      3.1.8 Extended Complex Number Object............ 22
      3.1.9 Array Object.............................. 23
      3.1.10 Linked Array Object...................... 23
      3.1.11 Character String Object.................. 25
      3.1.12 Hex String Object........................ 25
      3.1.13 Character Object......................... 25
      3.1.14 Unit Object.............................. 26
      3.1.15 Code Object.............................. 26
      3.1.16 Primitive Code Object.................... 27
      3.1.17 Program Object........................... 27
      3.1.18 List Object.............................. 28
      3.1.19 Symbolic Object.......................... 28
      3.1.20 Directory Object......................... 29
      3.1.21 Graphics Object.......................... 29
   3.2 Terminology and Abbreviations................... 30

4. Binary Integers..................................... 32
   4.1 Built-in Binary Integers........................ 32
   4.2 Binary Integer Manipulation..................... 34
      4.2.1 Arithmetic Functions...................... 34
      4.2.2 Conversion Functions...................... 35

5. Character Constants................................. 36

6. Hex & Character Strings............................. 37

6.1 Character Strings.......................... 37
6.2 Hex Strings................................ 39

7. Real Numbers..................................... 41
7.1 Built-in Reals............................. 41
7.2 Real Number Functions...................... 41

8. Complex Numbers.................................. 46
8.1 Built-in Complex Numbers................... 46
8.2 Conversion Words........................... 46
8.3 Complex Functions.......................... 46

9. Arrays... 48

10. Composite Objects............................... 49

11. Tagged Objects.................................. 51

12. Unit Objects.................................... 52

13. Temporary Variables and Temporary Environments.................................... 54
13.1 Structure of the Temporary Environment Area...................................... 55
13.2 Named vs. Unnamed Temporary Variables....... 57
13.3 Provided Words for Temporary Variables...... 59
13.4 Coding Suggestions......................... 60

14. Checking Arguments.............................. 61
14.1 Number of Arguments........................ 62
14.2 Dispatching on Argument Type............... 63
14.3 Examples................................... 66

15. Loop Control Structures......................... 68
15.1 Indefinite Loops........................... 68
15.2 Definite Loops............................. 70
15.2.1 Provided Words..................... 70
15.2.2 Examples........................... 71

16. Error Generation & Trapping..................... 73
16.1 Trapping: ERRSET and ERRTRAP............... 73
16.2 Action of ERRJMP........................... 73
16.3 The Protection Word........................ 74
16.4 Error Words................................ 75

17. Test and Control................................ 76
17.1 Flags and Tests............................ 76
17.1.1 General Object Tests............... 77
17.1.2 Binary Integer Comparisons......... 78
17.1.3 Decimal Number Tests............... 79
17.2 Words that Operate on the Runstream......... 80
17.3 If/Then/Else............................... 83
17.4 CASE words................................ 84

18. Stack Operations................................ 87

19. Memory Operations............................... 89
19.1 Temporary Memory........................... 89

19.2 Variables and Directories.................. 89
19.2.1 Directories....................... 91
19.3 The Hidden Directory....................... 92
19.4 Additional Memory Utilities................ 93

20. Display Management & Graphics.................... 94
20.1 Display Organization....................... 94
20.2 Preparing the Display...................... 95
20.3 Controlling Display Refresh................ 96
20.4 Clearing the Display....................... 97
20.5 Annunciator Control........................ 97
20.6 Display Coordinates........................ 98
20.6.1 Window Coordinates................ 98
20.7 Displaying Text............................ 99
20.7.1 Standard Text Display Areas........ 99
20.7.2 Temporary Messages................ 101
20.8 Graphics Objects........................... 102
20.8.1 Warnings.......................... 102
20.8.2 Graphics Tools.................... 103
20.8.3 Grob Dimensions................... 104
20.8.4 Built-in Grobs.................... 104
20.8.5 Menu Display Utilities............ 105
20.9 Scrolling the Display...................... 105

21. Keyboard Control................................. 109
21.1 Key Locations.............................. 109
21.2 Waiting for a Key.......................... 110
21.3 InputLine.................................. 111
21.3.1 InputLine Example................. 112
21.4 The Parameterized Outer Loop............... 113
21.4.1 The Parameterized Outer Loop Utilities........................ 114
21.4.2 Overview of the Parameterized Outer Loop....................... 115
21.4.3 Handling Errors with the Utilities......................... 116
21.4.4 The Display....................... 116
21.4.5 Error Handling.................... 117
21.4.6 Hard Key Assignments.............. 117
21.4.7 Menu Key Assignments.............. 119
21.4.8 Preventing Suspended Environments...................... 120
21.4.9 Specifying an Exit Condition....... 120
21.4.10 ParOuterLoop Example.............. 121

22. System Commands................................. 123

RPL PROGRAMMING GUIDE

1. Introduction

The HP 48 calculator was designed to be a customizable mathematical scratchpad for use by students and professionals in technical fields. In many respects it is a descendent of the HP 41, providing a much broader and more sophisticated computation capability than the HP 41, but preserving its RPN/key-per-function orientation.

The HP 48 uses the so-called Saturn architecture, named by the code name of the original CPU designed for the HP 71B handheld computer. It also uses a custom operating system/language called RPL, which was designed to provide symbolic mathematical capabilities, executing from ROM in a limited RAM environment (it is today still the only symbolic system that can run in ROM). The combination of specialized hardware and firmware makes it relatively difficult to develop application software for the HP48, and accordingly the HP48 is not positioned as a primary external application vehicle. The orientation of the product and its user programming language is towards simple customization by the primary user.

Despite these barriers, the price and physical configuration of the HP48 make it a desirable application platform for many software developers, especially those who want to target customers in the HP48's normal markets. The user language is suitable for simple programs, but for elaborate systems, the intentional error protection and other overhead can result in substantial performance penalties compared with the programs using the full range of system calls.

In this document, we will provide a description of the design and conventions of the RPL language. The material here should provide enough detail to permit the creation of RPL programs and other objects, using the associated IBM PC-based compilation tools. Included is documention of a large number of system RPL objects that are useful utilities for program development.

2. RPL Principles

(The following material was excerpted from "RPL: A Mathematical Control Language", by W. C. Wickes, published in "Programming Environments", Institute for Applied Forth Research, Inc., 1988)

2.1 Origins

In 1984, a project was started at Hewlett-Packard Corvallis Division to develop a new software operating system to streamline calculator development and support a new generation of hardware and software. Previously, all HP calculators were implemented entirely in assembly language, a process that was becoming increasingly cumbersome and inefficient as the memory sizes of the calculators increased. The objectives for the new operating system were as follows:

+ To provide execution control and memory management, including plug-in memory;
+ To provide a programming language for rapid prototyping and application development;
+ To support a variety of business and technical calculators;
+ To execute identically out of RAM and ROM;
+ To minimize memory use, especially RAM;
+ To be transportable to various CPU's;
+ To be extensible; and
+ To support symbolic mathematical operations.

Several existing operating systems and languages were considered, but none could meet all of the design objectives. A new system was therefore developed, which merges the threaded interpretation of Forth with the functional approach of Lisp. The resulting operating system, known unofficially as RPL (for Reverse-Polish Lisp), made its first public appearance in June of 1986 in the HP-18C Business Consultant calculator. Subsequently, RPL has been the basis for the HP-17B, HP-19B, HP-27S, HP-28C and HP-28S, and HP 48S and HP 48SX calculators. The HP-17B, 18C, and 19B are designed for business applications; they and the HP-27S scientific calculator offer an ``algebraic'' calculating logic, and the underlying operating system is invisible to the user. The HP 28/HP 48 families of scientific calculators use an RPN logic, and many of the facilities of operating system are directly available as calculator commands.

2.2 Mathematical Control

The official operating system objectives listed above were blended throughout the RPL development cycle with a less formal objective of creating a mathematical control language that would extend the ease-of-use and interactive nature of a calculator to the realm of symbolic mathematical operations. A calculator is distinguished from a computer in this context by:

+ very compact size;

+ ``instant on''--no warm-up or software loading/bootstrapping;

+ dedicated keys for common operations rather than qwerty keyboards.

+ ``instant action'' when a function key is pressed.

The HP-28, which was developed by the same team that created the RPL operating system, was the first realization of this background objective; the HP 48 is the latest and most mature implementation.

Much of the design of RPL can be derived from a consideration of the manner in which ordinary mathematical expressions are evaluated. Consider, for example, the expression

$$1+ 2 \sin(3x) +4$$

As any RPN enthusiast knows, the expression as written here does not correspond in its left-to-right order to the order in which a human or a machine could actually carry out the calculation. For example, the first sum has to be delayed until after several other steps are executed. Rewriting the expression in RPN form, we obtain a representation that is also executable in its written order:

```
1   2   3   x   *   sin   *   +   4   +
```

To translate this sequence into a control language, we need to formalize several concepts. First, we use the generic term object to refer to each step in the sequence, such as 1, 2, or sin. Even in this simple example, there are three classes of objects:

1. Data objects. Execution of an object such as 1, 2, or 3 in the example just returns the value of the object.

2. Names. The symbol x must be the name of some other object; when x is executed, the named object is substituted for the symbol.

3. Procedures. Objects such as *, sin, and + represent mathematical operations, which are applied, for example, to data objects to create new data objects.

The concept of an object is closely tied to the concept of execution, which can be thought of as the "activation" of an object. An individual object is characterized by its object type, which determines its action when executed, and its value, which distinguishes it from another of the same type.

Expression evaluation in these terms becomes the sequential execution of a series of objects (the objects representing the RPN form of the expression). Two constructs are necessary to make the execution coherent: an object stack and an interpreter pointer. The first construct provides a place from which procedure objects can take their arguments and to which they can return their result objects. A LIFO stack as used in Forth is ideal for this purpose, and such a stack is included in RPL. The interpreter pointer is just a program counter that indicates the next object to be executed. The interpreter pointer should be distinguished from the CPU program counter, which indicates the next CPU instruction.

A mathematical expression considered as a sequence of objects suggests an additional classification of objects as either atomic or composite. An atomic objects is an object that cannot be taken apart into stand-alone objects; examples are a simple data object like 1 or 2, or perhaps an object like * or + that is implemented normally in assembly language. A composite object is a collection of other objects. In Forth, a secondary word is an example of a composite object. RPL provides at least three types of composite objects: secondaries, which are prcedures defined as unrestricted sequences of objects; symbolics, which are sequences of objects that must be logically equivalent to algebraic expressions; and lists, which contain objects collected for any logical purpose other than sequential execution.

The final point in this brief mathematics-to-RPL derivation is the observation that the definition of composite objects leads to the concepts of threaded interpretation and a return stack. That is, in the example it it easy to imagine that the name object x could represent a composite object that in turn represents another expression. In that case, one would expect execution of x to cause the intepreter pointer to jump to the sequence of objects referenced by x, while the location of the object following x in the original is stored so that execution can later return there. This process should be able to be indefinitely repeated, so RPL provides a LIFO stack for the return objects.

The preceding introduction might in some respects also have been an introduction for the derivation of Forth, if questions of floating-point versus integer arithmetic are

ignored. In particular, both systems use threaded interpretation and a LIFO data stack for interchange of objects. However, there are several important differences between Forth and RPL:

+ RPL supports both direct and indirect threaded execution in a completely uniform manner.

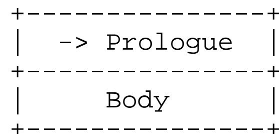
+ RPL supports dynamic allocation of its objects.

+ RPL code is, in general, completely relocatable.

2.3 Formal Definitions

This section will present the abstract definitions of RPL that are independent of any particular CPU or implementation.

The fundamental structure in RPL is the object. Any object consists of a pair: the prologue address and the object body.



```
+---------------+
|  -> Prologue  |
+---------------+
|     Body      |
+---------------+
```

The two parts are contiguous in memory with the prologue address part in lower memory The prologue address is that of a machine-code routine that executes the object; the body is data used by the prologue. Objects are classified by type; each type is associated with a unique prologue. The prologues thus serve a dual purposes of executing an object and identifying its type.

An object is either atomic or composite. A composite object is either null or non-null; a non-null composite has a head which is an object and a tail which is composite.

In addition to being executed, all RPL objects can be copied, compared, embedded in composite objects, and skipped. The latter property implies that the memory length of any object is predetermined or can be computed from the object. For atomic objects such as real numbers, the size is invariant. For a more complicated atomic object such as a numerical array, the size can be computed from the array dimensions that are stored in the body of the array object. (RPL arrays are not composite--the elements do not have individual prologues and hence are not objects.) Composite objects may include a length field or they may end with a marker object.

A pointer is an address in the memory space of the CPU, and may be a location pointer or an object pointer. A location pointer addresses any part of memory, whereas a object pointer point to an object, specifically to the prologue location pointer at the start of an object.

RPL requires, in addition to the CPU program counters, five variables for its fundamental operation:

+ The interpreter pointer I.

+ The current object pointer O.

+ The data stack pointer D.

+ The return stack pointer R.

+ The amount of free memory M.

In the most general definition of RPL, I is an object pointer pointing to a composite object that is the top of a stack of composite objects called the runstream. R points to the rest of the runstream stack. In practical implementations, this definition is streamlined by allowing I to point to any object embedded in a composite, while R is a location pointer pointing to the top of a stack of object pointers, each of which points to an embedded object.

It is fundamental to RPL that objects can be executed directly or indirectly with equivalent results. This means that an object can be represented anywhere by a pointer to the object as well as by the object itself.

2.4 Execution

RPL object execution consists of the CPU execution of the object's prologue, where the prologue code can access the object's body by means of the object pointer O. Object pointer execution is the CPU execution of the pointer's addressee. This interpretive execution is controlled by the inner interpreter, or inner loop, which determines the sequence of object/object pointer execution.

RPL objects are sorted by their general execution properties into three classes:

* Objects that merely return themselves to the data stack when executed are called data class objects. Examples are real numbers, strings, and arrays.

* Objects that serve as references for other objects are called identifier class objects. RPL defines three identifier class object types: identifier (global name), temporary identifier (local name), and ROM pointer (XLIB name).

* Objects that contain bodies into which execution flow can pass are called procedure class objects. There are three types of procedure class objects: programs (also called a "secondaries" or a "colon-definitions" in Forth terminology), code objects, and primitive code objects.

The RPL inner loop and prologue designs provide for interchangeable direct and indirect object execution (note: a patent application has been filed for the concepts described next). The inner loop consists of the following pseudo-code:

```
O = [I]
I = I + delta
PC = [O] + delta
```

where [x] means the contents of address x, and delta is the length of a memory address. This loop is the same in Forth, except that the CPU execution jumps to [O]+delta instead of to [O]. This is because all RPL prologues start with their own address, which is the feature that makes possible direct execution as well as indirect. Prologues look like this:

```
PROLOG  ->PROLOG                          Self address
        IF O + delta != PC THEN GOTO REST   Test for direct execution
        O = I - delta                       Correct O
        I = I + len                         Correct I
REST    (rest of prologue)
```

Here len is the length of the object body.

When an object is being executed directly, the inner loop does not set O or I correctly. However, a prologue knows it is being executed directly by comparing the PC address with O and can update the variables accordingly. A prologue is also responsible for preserving the threaded interpretation by including a return to the inner loop at its end.

This flexible interpretation is intrinsically slower than the indirect-only execution (like Forth), because of the overhead of making the direct/indirect test. In practical implementations of RPL, it is possible to shift the overhead almost entirely to the direct execution case, so that the execution penalty for the indirect case is negligible, including primitive assembly language objects that are never executed directly. The trick is to replace the last step of the inner loop with the Forth-like PC = [O], and, for prologues of directly-executable objects, replace the self-address at the start of each prologue with a slice of executable code delta in length. The compiled opcodes of this slice must also be the address of a meta-prologue that handles the direct execution case. In Saturn CPU implementations, the code slice consists of the instruction M=M-1 (decrementing available memory is common to virtually all directly-executable object prologues) plus a NOP instruction to fill out the delta length.

The virtue of direct execution is that it enables the straightforward management of nameless objects that are created during execution. During the course of symbolic algebraic manipulations, it is common to create, use, and discard any number of temporary intermediate results; the

necessity to compile and store these objects with some form of name for indirect reference, then uncompile them to recover memory, would make the whole process unmanageable. In RPL such objects can be placed on the stack, copied, embedded in composite objects, executed, and deleted. For example, a composite object representing the expression x + y can be added to a second object representing 2z, returning the result object x + y + 2z; furthermore, any of these objects could be embedded in a program object to perform the addition repetitively.

Although RPL is primarily a syntax-less postfix language in which procedures take their arguments from the stack and return results to the stack, it does provide operations that work on the runstream to provide for prefix operations and for alterations to the normal threaded execution. Foremost among the runstream operations is the quoting operation that takes the next object from the runstream and pushes it on the data stack to postpone its execution. This operation is similar in purpose to the Lisp QUOTE, but takes its RPL name ' (tick), from its Forth equivalent. RPL also has operations to push and pop objects from the return stack. (DO loop parameters, however, are not stored on the return stack, using a special environment instead.)

2.4.1 EVAL An object on the data stack may be indirectly executed by means of the RPL word EVAL, which pops an object from the stack and executes its prolog. The system object EVAL should be distinguished from the user RPL command EVAL. The latter is equivalent to system EVAL except for lists, symbolic objects, and tagged objects. For a tagged object, user EVAL executes the object contained in the body of the tagged object. For lists and symbolics, user EVAL dispatches to the system word COMPEVAL, which executes the object as if it were a program (see below).

2.4.2 Data_Class_Objects Object types in this class are:

Binary Integer Object (note: the user RPL binary integer is actually a hex string object in system RPL terms.)
Real Object
Extended Real Object
Complex Object
Extended Complex Object
Array Object
Linked Array Object
Character String Object
Hex String Object
Character Object
Graphics Object
Unit Object
List Object
Symbolic Object ("algebraic object")
Library Data Object
Directory Object
Tagged Object
External Object

All objects in the data class have the property that, when executed, they simply place themselves on the top of the data stack.

2.4.3 Identifier_Class_Objects Object types in this class are:

ROM Pointer Object (XLIB name)
Identifier Object (global name)
Temporary Identifier Object (local name)

Objects in the identifier class share the property that they serve to provide references for other objects. Identifier objects represent the resolution of global variables, and ROM Pointer Objects represent the resolution of commands stored in libraries. Temporary identifier objects, on the other hand, provide references for temporary objects in temporary environments.

Execution of a ROM pointer object (by the DOROMP prologue) entails locating and then executing the referenced ROM-WORD object part. Non-location is an error condition.

Execution of an identifier object (by the DOIDNT prologue) entails locating and then executing the referenced global variable object part. Non-location returns the identifier object itself.

Execution of a temporary identifier object (by the DOLAM prologue), entails locating the referenced temporary object and pushing it on the data stack. Non-location is an error condition.

2.4.4 Procedure_Class_Objects Object types in this class are:

Code Object
Primitive Code Object
Program Object

Objects in the procedure class share the property of executability, that is, executing a procedure class object involves passing control to executable procedure or code associated with the object.

Code objects contain assembly language sequences for direct execution by the CPU, but are otherwise normal, relocatable objects. Primitive code objects have no prolog in the usual sense; the prolog address field points directly to the object body, which contains an assembly language sequence. These objects can only exist in permanent ROM, and can never be executed directly. When a code object or primitive code object is executed, control is passed (by setting the PC) to the machine language instruction set contained in the object body. For a primitive code object, this control passing is done by the execution mechanism (EVAL or the inner loop) itself. For a code object, the prologue passes control by placing the PC at the beginning of the machine language slice contained in the object body. Note again that a primitive code object prologue (which is its body) need not contain logic to test for direct verses indirect execution (nor contain code to update I or O) since, by definition, it is never executed directly.

Execution of a program is sequential execution of the objects and object pointers that comprise the body of the program. The execution is threaded in that the objects in a program may themselves be secondaries or pointers to secondaries. When encountered by the inner loop, an embedded program is executed prior to resumption of execution of the current one.

The end of a program is marked by the object SEMI (from "semicolon"--a ";" is the closing delimiter recognized by the RPL compiler to mark the end of a program definition). Execution of SEMI pops the top object pointer from the return stack and resumes execution at that point.

2.4.5 Object_Skipover_and_SEMI One of the basic premises of RPL is that any RPL object that can be directly executed (which includes all object types except primitive code objects) must be traversable, that is, must have a structure which allows it to be skipped over. Object skipover occurs throughout the RPL system but most notably during direct execution by the inner loop when the interpreter pointer I must be set to point to the next object after the one being directly executed.

There exist both RPL objects and utilities to perform this object skipover function. In addition, objects are required to skipover themselves when being executed directly. The skipover mechanism for atomic objects is simple and straightforward since the object length is either known or is easily computable. For the composite objects (program, list, unit, symbolic) the length is not easily computable and the skipover function here is somewhat more involved, using an implicit recursion. These composite objects do not carry known or easily computable length information and therefore must have a tail delimiter, namely an object pointer to the primitive code object SEMI. Note that SEMI serves an explicit function for the program object (the procedure class composite object); for data class composite objects (list, unit, and symbolic objects), it only serves as a tail delimiter.

2.4.6 RPL_Pointers A pointer is defined to be an address and may be either a location pointer or an object pointer. A location pointer addresses any segment of the memory map while an object pointer specifically addresses an object. Note that, for example, the prologue address part of an object is a location pointer.

2.5 Memory Management

The uniformity of direct and indirect execution means not only that objects as well as object pointers can be embedded in the execution stream, but also that object pointers can logically replace objects. In particular, the RPL data and return stacks explicitly are stacks of object pointers. This means, for example, that an object on the data stack can be copied (e.g. by DUP) at a cost of only delta bytes of memory, regardless of the size of the object. Furthermore, duplication and similar stack operations are very fast.

Of course, the objects referenced on the stacks must exist somewhere in memory. Many, including all of the system objects that provide system management and an application language, are defined in ROM and can be referenced by a pointer with no other housekeeping implications. Objects created in RAM may exist in two places. Those that are unnamed are stored in a temporary object area, where each is maintained as long as it is referenced by a pointer anywhere in the system (this implies that if a temporary object moves, all pointers to it must be updated). Naming an object consists of storing it as a pair with a name field in a linked-list called the user object area. These objects are maintained indefinitely, until they are explicitly purged or replaced. A named object is accessed by means of an identifier object, which consists of an object with a name field as its body. Executing an identifier causes the user object area to be searched for an object stored with the same name, which is then executed. This run-time

resolution is intrinsically slower than the compile-time resolution used for ROM objects, but it allows for a dynamic and flexible system where the order in which objects are compiled is immaterial.

The process of naming objects by storing them with names in the user object area is augmented by the existence of local environments, in which objects can be bound to names (lambda variables) that are local to a currently executing procedure. The binding is abandoned when the procedure completes execution. This feature simpifies complicated stack manipulations by allowing the stack objects to be named and then referenced by name within the scope of a defining procedure.

RPL provides that any object stored in the user object area can be deleted without corrupting anything in the system. This requires certain design conventions:

+ When a RAM object is stored in the user object area, a new copy of the object is stored, not a pointer to the object.

+ Pointers to RAM objects are not permitted in composite objects. When a composite object is created from stack objects, RAM objects are copied and directly embedded in the composite. When a stored object is represented by name in a composite, it is the identifier object that is embedded, not a location pointer as in Forth.

+ If a stored object is referenced by any pointers on the stacks at the time when it is purged, it is copied to the temporary object area and the pointers to it are updated accordingly. This means that the memory associated with an object is not recovered until the last reference to it is deleted.

The use of temporary objects with multiple references means that a temporary object can not necessarily be deleted from memory immediately when a single reference to it is eliminated. In current RPL implementations, no memory recovery at all is performed until the system runs out of memory (M=0), at which time all unreferenced objects in the temporary object area are deleted. The process, called "garbage collection" can be significantly time-consuming, so that RPL execution does not proceed uniformly.

From the preceding discussion, it will be apparent that RPL is not as fast in general as Forth because of its extra interpretation overhead and greatly elaborated memory management scheme. While maximum execution speed is always desirable, the design of RPL emphasizes its role as an interactive mathematical control language in which flexibility, ease of use, and the ability to manipulate procedural information are paramount. In many cases, these attributes of RPL result in faster problem-solving

throughput than Forth, which executes faster but is more difficult to program.

RPL also provides for objects that are intermediate between those fixed in ROM and those that are mobile in RAM. A library is a collection of objects, organized in a permanent structure that permits parse-time and run-time resolution by means of tables included in the library. An XLIB name is an identifier class object that contains a library number and an object number within the library. Execution of an XLIB name executes the stored object. The identities and locations of libraries are determined at system configuration. A particular library can be associated with its own RAM directory, so that, for example, a library might contain permanent formulas for which the variable values are maintained in RAM.

2.6 User RPL and System RPL

There is no fundamental difference between the HP 48 programming language, which we will call "user RPL," and the "system RPL" in which HP 48 functionality is implemented. User language programs are executed by the same inner loop interpreter as system programs, with the same return stack. The data stack displayed on the HP 48 is the same as that used by system programs. The distinction between user RPL and system RPL is only one of scope: user RPL is a subset of system RPL. User RPL does not provide direct access to all of the data class object types that are available; the use of built-in procedures is limited to those that are provided as commands.

A "command" is procedure-class object stored in a library, along with a text string that serves as the command's name. The name is used for compiling and decompiling the object. When the command line parser matches text in the command line with a command name, it compiles an object pointer if the command is contained in a library in the HP 48's permanent ROM. Otherwise it compiles the corresponding XLIB name. Also, built-in command objects are preceded in ROM by a six-nibble field that is the body of an XLIB name. When the decompiler encounters an object pointer, it looks for this field in the ROM ahead of the object; if it finds a valid field, it then uses the information there to locate a text command name to display. Otherwise, it decompiles the object itself.

Commands are distinguished from other procedure objects by certain conventions in their design. Structurally, all commands are program objects, the first object in which is one of the system dispatch objects CK0, CK1&Dispatch, CK2&Dispatch, CK3&Dispatch, CK4&Dispatch, and CK5&Dispatch (see section 13). CK0, which is used by zero-argument commands, may be followed by any additional objects. CK1&Dispatch ... CK&Dispatch must be followed by a sequence

of pairs of objects; the first of each pair identifies a stack argument type combination, and the second specifies the object to execute for each corresponding combination. The last pair is followed by the end-program marker object (SEMI).

The other command object conventions govern their behavior. In particular, they should:

* remove any temporary objects from the stack, returning only the specified results;

* do any range checking necessary to ensure that errors do not occur that might cause disasters;

* restore HP48 modes to their original states, unless the command is specifically for changing a mode.

The overhead involved in these structure and behavior conventions does impose a minor performance penalty. However, the primary execution speed advantage of system RPL over user RPL comes simply from the larger set of available procedures in system RPL, access to fast binary arithmetic, and improved control over system resources and execution flow.

2.7 Programming in System RPL

Writing programs in system RPL is no different in principle than in user RPL; the difference lies in the syntax and scope of the compiler. For user RPL, the compiler is the command line ENTER, the logic of which is documented in the owners' manuals. For system RPL developed on a PC, the compiler has several parts. The immediate analog of the command line parser is the program RPLCOMP, which parses source code text into Saturn assembly language. (The syntax used by RPLCOMP is described in xxx.) The output of RPLCOMP is passed to the assembler program SASM, which produces assembled object code. The program SLOAD resolves symbol references in SASM's output, finally returning executable code suitable for downloading into the HP 48. Individual objects can be collected in an HP 48 directory that is transferred back to the PC, where the program USRLIB can transform the directory into a library object. (It would be desirable to create a library directly on the PC, but the program to do this is not available at present.)

For the purpose of illustration, consider a hypothetical project development process that will result in a library object constructed with the USRLIB tool. The library is to contain a single command, BASKET, which calculates basket weaving factors according to several input parameters. BASKET should be designed with the structure described above for commands. In addition, assume that BASKET calls several other programs which are not to be user-accessible. To

achieve this, the objects are compiled on the PC, then downloaded into the HP 48 in a common directory, stored as BASKET, B1, B2, ... , where the latter variables contain the subroutines. The directory is uploaded to the PC, where USRLIB is applied to it with the directive that B1, B2, ... are to be "hidden."

There is no requirement that a program produced with the RPL compiler must be presented in a library object - if the entire application can be written within a single program, then so much the better. As programs grow beyond some reasonable level of complexity, this becomes more difficult, and a library object approach with multiple variables becomes easier to manage.

1. Create the source file on the PC using your favorite editor. The program source file name should have a ".s" extension, such as "prog.s". Use the compiler RPLCOMP.EXE to produce the Saturn assembler source file "prog.a".

2. Use the Saturn assembler SASM.EXE to assemble the program and produce an output file "prog.o".

3. Use the Saturn loader SLOAD.EXE to resolve your program's calls to HP 48 operating system. SLOAD.EXE output files may have any name, but the ".ol" extension is often used.

4. Download the final file (use binary transfer!) to the HP 48, and try out your code.

5. Upload the directory containing one or more objects to the PC, and use USRLIB.EXE to convert it to a library.

2.8 Sample RPL Program

To get acquainted with the process of producing a program written in internal RPL, consider the following example, which we'll call TOSET.

2.8.1 The_Source_File

This program removes duplicate objects from a list by decomposing the list into a series of objects on the stack, creating a new empty list, and putting the stack objects into the new list if they're unique.

```
* ( {list} --> {list}' )
ASSEMBLE
        NIBASC  /HPHP48-D/
RPL
::
  CK1NOLASTWD                   ( *Req. 1 argument* )
  CK&DISPATCH0 list
    ::
      DUPNULL{}? ?SEMI          ( *Exit for empty list* )
      INNERCOMP                 ( objn ... obj1 #n )
      reversym                  ( obj1 ... objn #n )
      NULL{} SWAP               ( obj1 ... objn {} #n )
      ZERO_DO (DO)
        SWAP                    ( obj1 ... objn-1 {} objn )
        apndvarlst              ( obj1 ... objn-1 {}' )
      LOOP
   ;
;
```

The first line is a comment, showing the input and output conditions for the program. Comments are denoted by an asterisk (*) in the first column, or within parentheses. Every programmer has their own style for comments. The style shown here is that objects are shown with stack level one on the right. Text is enclosed in asterisks.

The sequence

```
ASSEMBLE
        NIBASC  /HPHP48-D/
RPL
```

is a command to the assembler that includes the header for binary data transfer from the PC to the HP48. This is included here for simplicity, but could be included from another file by the loader.

The first command, CK1NOLASTWD, requires the stack contain at least one item, and clears the ram location which stores the name of the current command. This is important, because

you don't want to attribute errors encountered in this program to the last function that generated an error.

The second command, CK&DISPATCH0, reads a structure of the form

```
        type action
        type action
        ...
```

to decide what action to take based on the TYPE of object presented. If the type of object in level 1 does not have an entry in the table, the error "Bad Argument Type" will be generated. In this example, only one type of argument, a list, is acceptable, and the corresponding action is a secondary. For more on argument checking commands, see the chapter "Argument Validation".

The command DUPNULL{}? returns the list and a TRUE/FALSE flag which indicates if the list is empty. The command ?SEMI exits the secondary if the flag is TRUE.

The command INNERCOMP is an internal form of the user word LIST->. The number of objects is returned in level one as a binary integer (see the chapter "Binary Integers").

The command "reversym" reverses the order of #n objects on the stack. This is used here to account for the ordering of objects placed in a list by the "apndvarlst" which is described below.

The ZERO_DO command begins a counted loop. This loop will process each object in the original list. The (DO) command tells RPLCOMP that this is the start of a loop, otherwise the LOOP command would be flagged as unmatched.

The "apndvarlst" command appends an object to a list if and only if that object does not appear in the list already.

The LOOP command ends the loop. For more on loop commands, see the chapter "Loop Structures".

2.8.2 Compiling_the_Program To compile the program for the HP 48, follow these steps:

1. Store the example code in a file TOSET.S.

2. RPLCOMP TOSET.S TOSET.A

   This command compiles the RPL source and produces a Saturn assembler source file.

3. SASM TOSET.A

   This command assembles the Saturn source file to produce the files TOSET.L and TOSET.O.

4. The file TOSET.M is a loader control file that looks like this:

```
TITLE Example           <-- Specifies a listing title
OUTPUT TOSET            <-- Specifies the output file
LLIST TOSET.LR          <-- Specifies the listing file
SUPPRESS XREF           <-- Suppresses the cross ref
SEARCH ENTRIES.O        <-- Reads HP48 entries
REL TOSET.O             <-- Loads TOSET.o
END
```

   Create the file TOSET.M and invoke the loader:

   SLOAD -H TOSET.M

Check the file TOSET.LR for errors. An unresolved reference usually points to a misspelled command. Now download the file TOSET into the HP 48 and give it a try!

Enter the list { 1 2 2 3 3 3 4 }, evaluate TOSET, and you should get { 1 2 3 4 }.

3. Object Structures

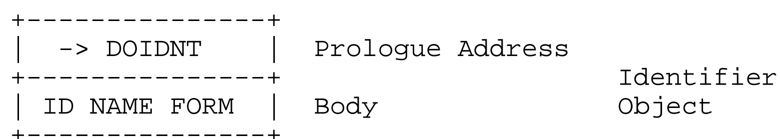
This chapter provides additional information about some of the RPL object types supported by the HP 48. Although the information is primarily relevant to assembly language programming, a knowledge of object structure can often help in understanding performance and efficiency issues in RPL programming.

Unless explicitly stated otherwise, all specifically-defined fields within an object body are assumed to be 5 nibbles, the CPU address width.

3.1 Object Types

3.1.1 Identifier_Object

An identifier object is atomic, has the prologue DOIDNT, and a body which is an ID Name form.

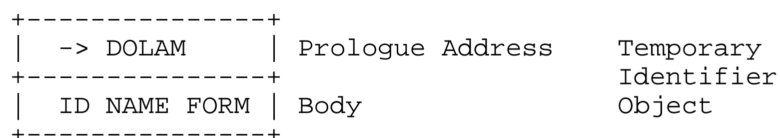


An ID name form is a character sequence preceded by a one-byte character count field.

Identifier objects are, among other things, the compiletime resolution of global variables.

3.1.2 Temporary_Identifier_Object

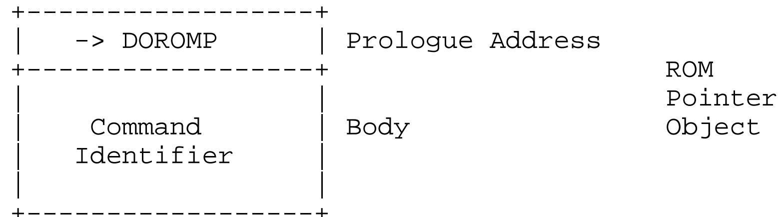
A temporary identifier object is atomic, has the prologue DOLAM, and a body which is an ID name form.



Temporary identifier objects provide named references for temporary objects bound to the identifiers in the formal parameter list of a temporary variable structure.

3.1.3 ROM_Pointer_Object

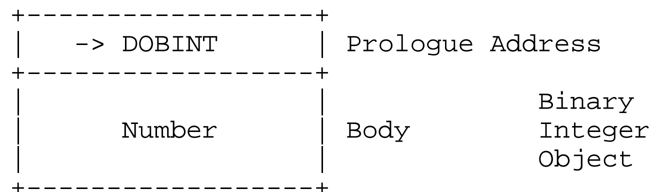
A ROM pointer object, or XLIB name, is atomic, has the prologue DOROMP, and a body which is a ROM-WORD identifier.



ROM pointer objects are the compiletime resolution of commands in mobile libraries. A command indentifier identifier is a pair of 12 bit fields: the first field is a library ID number, and the second field is the command ID number within the library.

3.1.4 Binary_Integer_Object

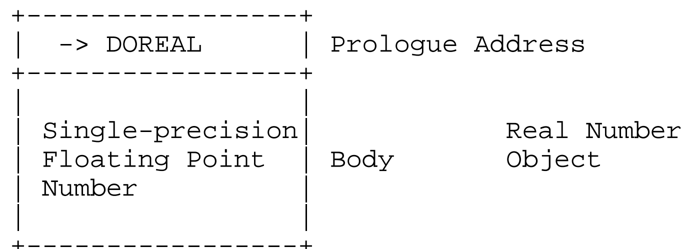
A binary integer object is atomic, has the prologue DOBINT, and a body which is a 5-nibble number.



The use of this object type is to represent binary integers whose precision is equivalent to a memory address.

3.1.5 Real_Number_Object

A real number object is atomic, has the prologue DOREAL, and a body which is a single-precision floating point number (or real number, for short).



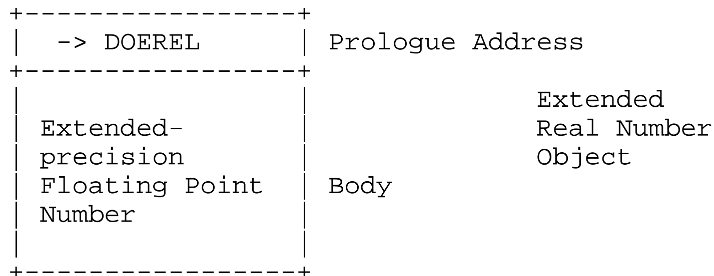
One use of this object type is to represent packed floating-point numbers (eight bytes) on a Saturn system and, in this application, the body of the object may consist of 16 BCD nibbles as follows:

```
(low mem)      EEEMMMMMMMMMMMMS
```

where S is the numeric sign (0 for nonnegative and 9 for negative), MMMMMMMMMMMM is a 12 digit mantissa with an implied decimal point between the first and second digits and the first digit nonzero if the number is nonzero, and EEE the exponent in tens complement form (-500 < EEE < 500).

3.1.6 Extended_Real_Number_Object

An extended real number object is atomic, has the prologue DOEREL, and a body which is an extended-precision floating point number (or extended real, for short).



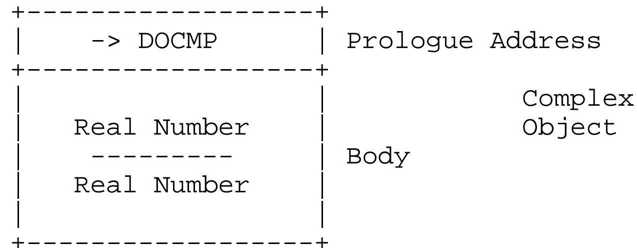
One use of this object type is to represent unpacked floating-point numbers (10.5 bytes) on a Saturn system and, in this application, the body of the object may consist of 21 BCD nibbles as follows:

```
(low mem)     EEEEEMMMMMMMMMMMMMMMS
```

where S is the numeric sign (0 for nonnegative, 9 for negative), MMMMMMMMMMMMMMM is a 15 digit mantissa with an implied decimal point between the first and second digits and the first digit nonzero if the number is nonzero, and EEEEE the exponent in tens complement form (-50000 < EEEEE < 50000).

3.1.7 Complex_Number_Object

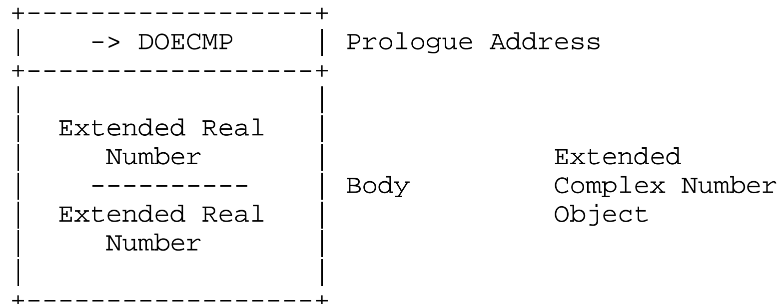
A complex number object is atomic, has the prologue DOCMP, and a body which is a pair of real numbers.



The use of this object type is to represent single-precision complex numbers, where the real part is interpreted as the first real number in the pair.

3.1.8 Extended_Complex_Number_Object

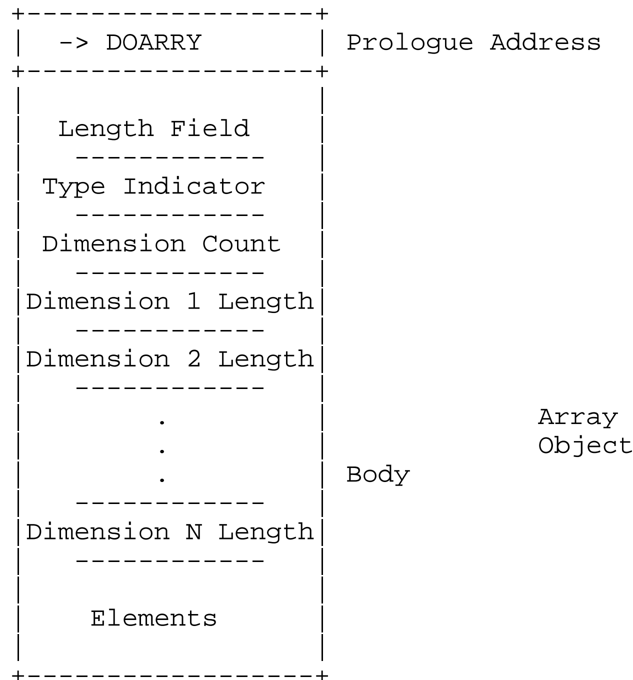
An extended complex number object is atomic, has the prologue DOECMP, and a body which is a pair of extended real numbers.



The use of this object type is to represent extended-precision complex numbers in the same way as for the complex object.

3.1.9 Array_Object

An array object is atomic, has the prologue DOARRY, and a body which is a collection of the array elements. The body also includes a length field (indicating the length of the body), a type indicator (indicating the object type of its elements), a dimension count field, and length fields for each dimension.



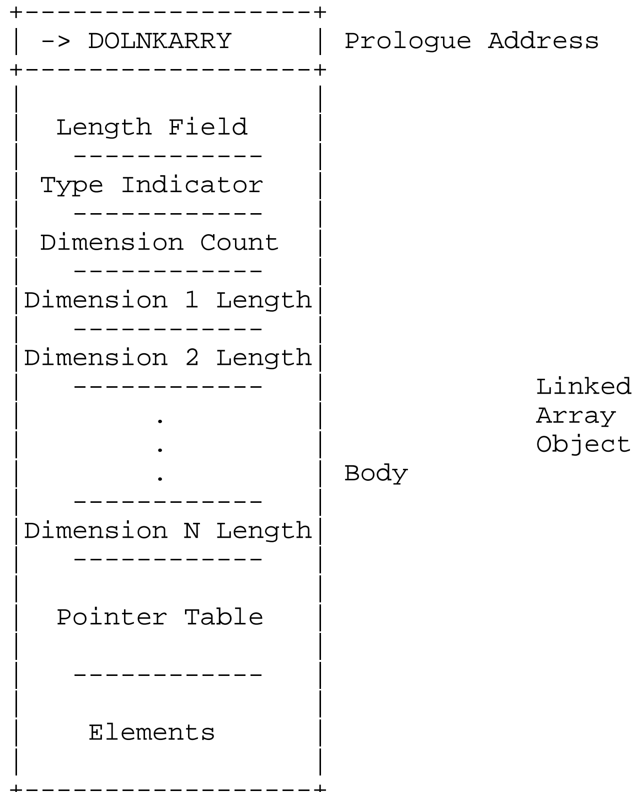
The array elements are object bodies of the same object type. The type indicator is a prologue address (think of this prologue address as applying to each element of the array).

Array "OPTION BASE" is always 1. A null array is designated by any dim limit having the value zero. All elements of an array object are always present as indicated by the dimensionality information and are ordered in memory by the lexicographic order of the array's indices.

3.1.10 Linked_Array_Object

A linked array object is atomic, has the prologue DOLNKARRY, and a body which is a collection of the array elements. The body also includes a length field (indicating the length of the body), a type indicator (indicating the object type of its elements), a dimension count field, length fields for each dimension, and a pointer table whose contents are forward self-relative offsets to the array elements; the elements of the pointer table are ordered in memory by the

lexicographic order of the array's indices.

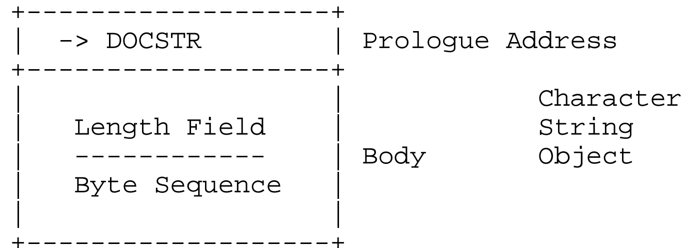


The array elements are object bodies of the same object type. The type indicator is a prologue address (think of this prologue address as applying to each element of the array).

Linked array "OPTION BASE" is always 1. A null linked array is designated by any dim limit having the value zero. There is no assumption on the ordering of the elements of a linked array object, nor on their presence; absence of an element lying on an allocated dimension is indicated by the value zero occupying the corresponding pointer table element.

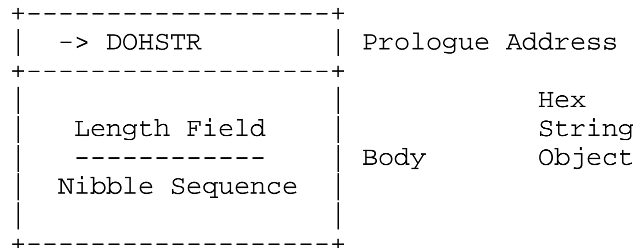
3.1.11 Character_String_Object

A character string object is atomic, has the prologue DOCSTR, and a body which is a character string (a byte sequence). The body also includes a length field (indicating the length of the body).



3.1.12 Hex_String_Object

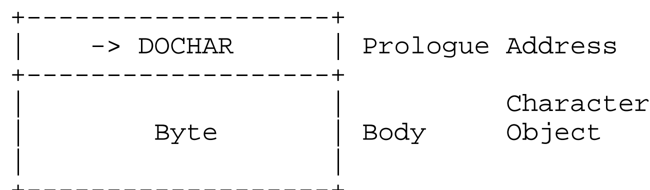
A hex string object is atomic, has the prologue DOHSTR, and a body which is a nibble sequence. The body also includes a length field (indicating the length of the body).



A typical use for this object type is a buffer or table. Hex string objects of 16 nibbles or fewer are used to represent user RPL binary integer objects.

3.1.13 Character_Object

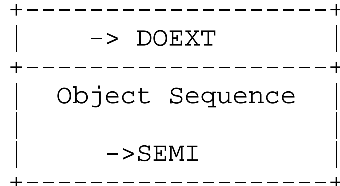
A character object is atomic, has the prologue DOCHAR, and a body which is a single byte.



This object type is used to represent one-byte quantities, such as ASCII or ROMAN8 characters.

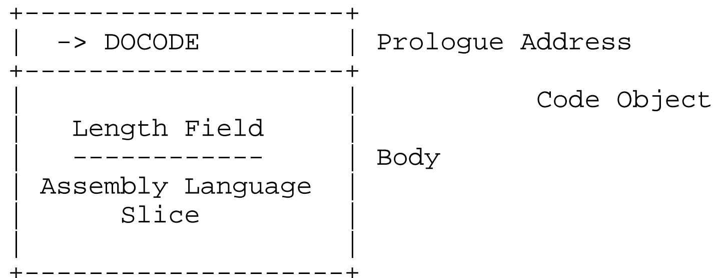
3.1.14 Unit_Object

A unit object is composite, has the prologue DOEXT, and a body which is a sequence consisting of a real number followed by unit name strings, prefix characters, unit operators, and real number powers, tail delimited by a pointer to SEMI.



3.1.15 Code_Object

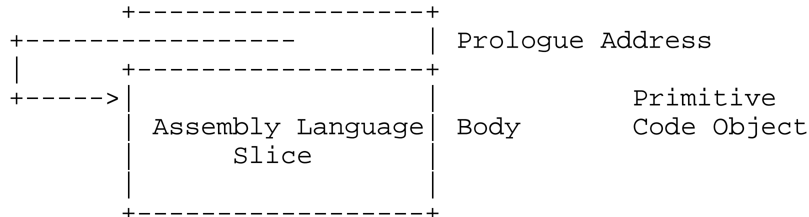
A code object is atomic, has the prologue DOCODE, and a body which is an assembly language slice. The body also includes a length field (indicating the length of the body). When executed, the prologue places the system program counter at the assembly language slice within the body.



The major applications for this object type are assembly language procedures which can be directly embedded in composite objects or exist in RAM.

3.1.16 Primitive_Code_Object

A primitive code object is a special case of a code object, used to represent code primitives in built-in libraries. The prologue of a primitive code object is its body, which is an assembly language slice; thus, when executed, the body executes itself.

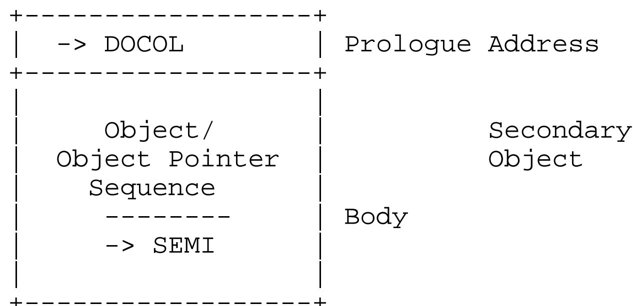


The primary purpose of this object type is more rapid execution of code objects in built-in libraries, that is, these objects are executed without the extra level inherent in separate prologue execution. However, their structure implies that (1) they can only exist in built-in libraries (never in RAM or mobile libraries) since the body must exist at a fixed address, (2) they cannot be skipped, and (3) they cannot exist in any situation where traversal may be required, such as an element of an array or an object within any composite object.

Note that this object type is an exception to the object type classification scheme presented at the beginning of this document. However, an object is a primitive code object if and only if the prologue address equals the object address plus 5. In addition, the prologues for this object type (that is, the object bodies) need not contain logic to test for direct verses indirect execution since, by definition, they cannot be executed directly.

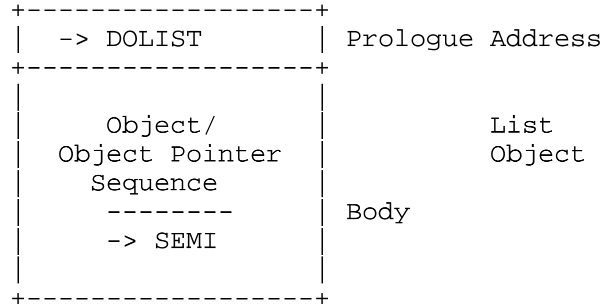
3.1.17 Program_Object

A program object (secondary) is composite, has the prologue DOCOL, and a body which is a sequence of objects and object pointers, the last of which is an object pointer whose pointee is the primitive code object SEMI.



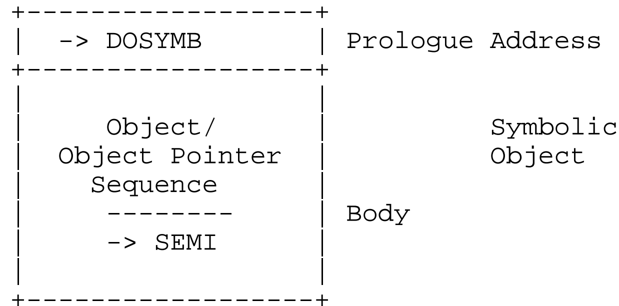
3.1.18 List_Object

A list object is composite, has the prologue DOLIST, and a body which is a sequence of objects and object pointers, the last of which is an object pointer whose pointee is the primitive code object SEMI.



3.1.19 Symbolic_Object

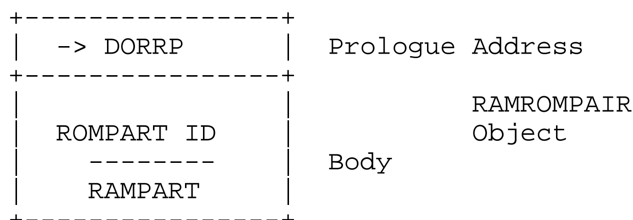
A symbolic object is composite, has the prologue DOSYMB, and a body which is a sequence of objects and object pointers, the last of which is an object pointer whose pointee is the primitive code object SEMI.



This object type is used to represent symbolic objects for symbolic math applications.

3.1.20 Directory_Object

A directory (RAMROMPAIR) object is atomic, has the prologue DORRP and a body which consists of a Library ID number and a RAMPART (linked list of variables--object/name pairs.



3.1.21 Graphics_Object

A graphics object is atomic, has the prologue DOGROB and a body which consists of the following:

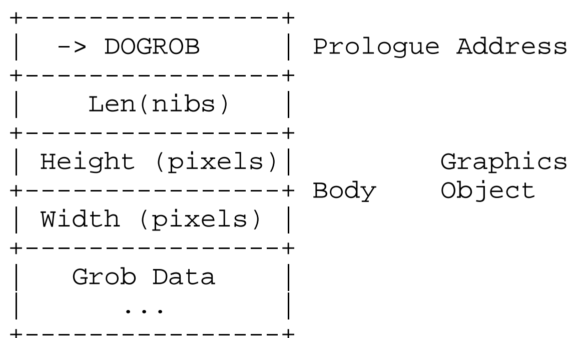
+ A 5 nibble length field for the data which follows.

+ A five nibble quantity that describes the height of the graphic in pixels.

+ A five nibble quantity that describes the width of the graphic in pixels.

+ The data.

The actual row dimension in nibbles (W) is always even for hardware reasons, hence each row of pixel data is padded with anywhere from 0-7 bits of wasted data.



The data nibbles begin at the upper-left corner of the graphics object and proceed left-to-right, top-to-bottom. Each row of pixel data is padded as needed to obtain an even number of nibbles per row. Thus the width in nibbles W is determined by:

W=CEIL(Width in pixels)/8

The bits in each nibble are written in reverse order, so the leftmost displayed pixel in a nibble is represented by the least-significant bit of the nibble.

3.2 Terminology and Abbreviations.

In the stack diagrams used throughout the remainder of this document, the following symbols are used to represent the various object types:

```
ob .......... Any object
id .......... Identifier Object
lam ......... Temporary Identifier Object
romptr ...... ROM Pointer Object
__# ........... Binary Integer Object
% ........... Real Object
%% .......... Extended Real Object
C% .......... Complex Object
C%% ......... Extended Complex Object
arry ........ Array Object
lnkarry ..... Linked Array Object
$ ........... Character String Object
hxs ......... Hex String Object
chr ......... Character Object
ext ......... External Object
code ........ Code Object
primcode .... Primitive Code Object
:: .......... Secondary Object
{} .......... List Object
symb ........ Symbolic Object
comp ........ Any Composite Object (list, secondary, symbolic)
rrp ......... Directory Object
tagged ...... Tagged Object
flag ........ TRUE/FALSE
```

(TRUE and FALSE above denote the object parts of built-in ROM-WORDs having these names. The addresses of these objects (that is, their data stack representations) are interpreted by RPL control structures as the appropriate truth value. Both objects are primitive code objects which, when executed, place themselves on the data stack).

In addition to the above notation, some additional terminology is useful.

ELEMENT:
An ELEMENT of a composite object is any object or object pointer in the body of the composite object.

CORE:

of a character string: the core of a character string object is the character data in the body.

of a hex string: the core of a hex string object is the nibble sequence in the body.

of a composite: the core of a composite object is the element sequence in the body not including the trailing object pointer to semi.

LENGTH:

of a character string: the length of a character string object is the number of characters in the core.

of a hex string: the length of a hex string object is the number of nibbles in the core.

of a composite: the length of a composite object is the number of elements in the core.

NULL:

character string: a null character string object is one whose length is zero.

hex string: a null hex string object is one whose length is zero.

composite: a null composite object is one whose length is zero.

INTERNAL:

an internal of a composite object is any object in the core of the composite object or the pointee of any object pointer in the core of the composite object.

(A composite object is often loosely referred to as containing a specific object type, for example "a list of binary integers"; what is meant is that the core internals are all of this object type).

4. Binary Integers

Internal binary integers have a fixed size of 20 bits, and are the most often used type for counting, loops, etc. Binary integers offer advantages of size and speed.

NOTE: User level binary integers are implemented as hex strings, so a user's object #247d is actually a hex string, and should not be confused with a binary integer whose prologue is DOBINT.

4.1 Built-in Binary Integers

The RPLCOMP compiler interprets a decimal number in a source file as a directive to produce a binary integer object - using a prologue and a body. Built-in binary integers can be accessed with just an object pointer. For instance, " 43 " (no quotes) in the source file produces a binary object:

```
        CON(5)  =DOBINT
        CON(5)  43
```

The object takes five bytes, but can be replaced by the word "FORTYTHREE", which is a supported entry point which would generate the following code:

```
        CON(5)  =FORTYTHREE
```

One pitfall to be aware of in binary integer naming conventions is the difference between the entries FORTYFIVE and FOURFIVE. In the former case, the value is decimal 45, but the latter is decimal 69. Names like 2EXT and IDREAL, where the values are not obvious, are used in conjunction with the CK&Dispatch family of argument checking commands. The names for the CK&Dispatch family are equated to the same places as other bints. This has been done for readability. For instance, the word SEVENTEEN, for decimal 17, has the names 2REAL and REALREAL equated to the same location. A trailing "d" or "h" on a name such as BINT_122d or BINT80h indicates the base associated with the value.

Words such as ONEONE, ZEROONE, etc. put more than one binary integer on the stack. These are indicated by a tiny stack diagram in parentheses, such as (--> #1 #1) for ONEONE.

The supported entries for binary integers are listed below
with the hex value in parentheses where needed:

```
2EXT (#EE)         FORTYNINE          SYMREAL (#A1)
2GROB (#CC)        FORTYONE           SYMSYM (#AA)
2LIST (#55)        FORTYSEVEN         TAGGEDANY (#D0)
2REAL (#11)        FORTYSIX           TEN
3REAL (#111)       FORTYTHREE         THIRTEEN
Attn# (#A03)       FORTYTWO           THIRTY
BINT253            FOUR               THIRTYEIGHT
BINT255d           FOURFIVE           THIRTYFIVE
BINT40h            FOURTEEN           THIRTYFOUR
BINT80h            FOURTHREE          THIRTYNINE
BINTC0h            FOURTWO            THIRTYONE
BINT_115d          FOURTY             THIRTYSEVEN
BINT_116d          IDREAL (#61)       THIRTYSIX
BINT_122d          INTEGER337         THIRTYTHREE
BINT_130d          LISTCMP (#52)      THIRTYTWO
BINT_131d          LISTLAM (#57)      THREE
BINT_65d           LISTREAL (#51)     TWELVE
BINT_91d           MINUSONE(#FFFFF)   TWENTY
BINT_96d           NINE               TWENTYEIGHT
Connecting(#C0A)   NINETEEN           TWENTYFIVE
EIGHT              ONE                TWENTYFOUR
EIGHTEEN           ONEHUNDRED         TWENTYNINE
EIGHTY             ONEONE(--> #1 #1)  TWENTYONE
EIGHTYONE          REALEXT (#1E)      TWENTYSEVEN
ELEVEN             REALOB (#10)       TWENTYSIX
EXT (#E)           REALOBOB (#100)    TWENTYTHREE
EXTOBOB (#E00)     REALREAL (#11)     TWENTYTWO
EXTREAL (#E1)      REALSYM (#1A)      TWO
EXTSYM  (#EA)      ROMPANY (#F0)      XHI
FIFTEEN            SEVEN              XHI-1 (#82)
FIFTY              SEVENTEEN          ZERO
FIFTYEIGHT         SEVENTY            ZEROZERO (--> #0 #0 )
FIFTYFIVE          SEVENTYFOUR        ZEROZEROONE (--> #0 #0 #1 )
FIFTYFOUR          SEVENTYNINE        ZEROZEROTWO (--> #0 #0 #2 )
FIFTYNINE          SIX                ZEROZEROZERO (--> #0 #0 #0 )
FIFTYONE           SIXTEEN            char (#6F)
FIFTYSEVEN         SIXTY              id (#6)
FIFTYSIX           SIXTYEIGHT         idnt (#6)
FIFTYTHREE         SIXTYFOUR          infreserr (#305)
FIFTYTWO           SIXTYONE           intrptderr (#a03)
FIVE               SIXTYTHREE         list (#5)
FIVEFOUR           SIXTYTWO           ofloerr (#303)
FIVESIX            SYMBUNIT (#9E)     real (#1)
FIVETHREE          SYMEXT (#AE)       seco (#8)
FORTY              SYMID (#A6)        str (#3)
FORTYEIGHT         SYMLAM (#A7)       sym (#A)
FORTYFIVE          SYMOB (#A0)        symb (#9)
FORTYFOUR
```

4.2 Binary Integer Manipulation

4.2.1 Arithmetic_Functions

```
#*               ( #2 #1 --> #2*#1 )
#+               ( #2 #1 --> #2+#1 )
#+-1             ( #2 #1 --> #2+#1-1 )
#-               ( #2 #1 --> #2-#1 )
#-#2/            ( #2 #1 --> (#2-#1)/2 )
#-+1             ( #2 #1 --> (#2-#1)+1 )
#/               ( #2 #1 --> #remainder #quotient )
#1+              ( # --> #+1 )
#1+'             ( # --> #+1 and quotes next runstream object
#1+DUP           ( # --> #+1 #+1 )
#1-              ( # --> #-1 )
#10*             ( # --> #*10 )
#10+             ( # --> #+10 )
#12+             ( # --> #+12 )
#2*              ( # --> #*2 )
#2+              ( # --> #+2 )
#2-              ( # --> #-2 )
#2/              ( # --> FLOOR(#/2) )
#3+              ( # --> #+3 )
#3-              ( # --> #-3 )
#4+              ( # --> #+4 )
#4-              ( # --> #-4 )
#5+              ( # --> #+5 )
#5-              ( # --> #-5 )
#6*              ( # --> #*6 )
#6+              ( # --> #+6 )
#7+              ( # --> #+7 )
#8*              ( # --> #*8 )
#8+              ( # --> #+8 )
#9+              ( # --> #+9 )
#MAX             ( #2 #1 --> MAX(#2,#1) )
#MIN             ( #2 #1 --> MIN(#2,#1) )
2DUP#+           ( #2 #1 --> #2 #1 #1+#2 )
DROP#1-          ( # ob --> #-1 )
DUP#1+           ( # --> # #+1 )
DUP#1-           ( # --> # #-1 )
DUP3PICK#+       ( #2 #1 --> #2 #1 #1+#2 )
OVER#+           ( #2 #1 --> #2 #1+#2 )
OVER#-           ( #2 #1 --> #2 #1-#2 )
ROT#+            ( #2 ob #1 --> ob #1+#2 )
ROT#+SWAP        ( #2 ob #1 --> #1+#2 ob )
ROT#-            ( #2 ob #1 --> ob #1-#2 )
ROT#1+           ( # ob ob' --> ob ob' #+1 )
ROT+SWAP         ( #2 ob #1 --> #1+#2 ob )
SWAP#-           ( #2 #1 --> #1-#2 )
SWAP#1+          ( # ob --> ob #+1 )
SWAP#1+SWAP      ( # ob --> #+1 ob )
SWAP#1-          ( # ob --> ob #-1 )
SWAP#1-SWAP      ( # ob --> #-1 ob )
SWAPOVER#-       ( #2 #1 --> #1 #2-#1 )
```

4.2.2 Conversion_Functions

```
COERCE          ( % --> # )  If %<0 then # is 0
                             If %>FFFFF then #=FFFFF
COERCE2         ( %2 %1 --> #2 #1 ) See COERCE
COERCEDUP       ( % --> # # ) See COERCE COERCESWAP      (
ob % --> # ob ) UNCOERCE        ( # --> % )
UNCOERCE%%      ( # --> %% ) UNCOERCE2       ( #2 #1 --> %2
%1 )
```

5. Character Constants

The following words are useful for converting between character objects and other object types:

```
CHR>#           ( chr --> # )
#>CHR           ( # --> chr )
CHR>$           ( chr --> $ )
```

The following character constants and strings are supported:

```
CHR_# CHR_* CHR_+ CHR_, CHR_- CHR_. CHR_/ CHR_0 CHR_1 CHR_2
CHR_3 CHR_4 CHR_5 CHR_6 CHR_7 CHR_8 CHR_9 CHR_: CHR_; CHR_<
CHR_= CHR_> CHR_A CHR_B CHR_C CHR_D CHR_E CHR_F CHR_G CHR_H
CHR_I CHR_J CHR_K CHR_L CHR_M CHR_N CHR_O CHR_P CHR_Q CHR_R
CHR_S CHR_T CHR_U CHR_V CHR_W CHR_X CHR_Y CHR_Z CHR_a CHR_b
CHR_c CHR_d CHR_e CHR_f CHR_g CHR_h CHR_i CHR_j CHR_k CHR_l
CHR_m CHR_n CHR_o CHR_p CHR_q CHR_r CHR_s CHR_t CHR_u CHR_v
CHR_w CHR_x CHR_y CHR_z

CHR_00 (hex 0) CHR_...  CHR_DblQuote    CHR_-> CHR_<<
CHR_>> CHR_Angle CHR_Deriv CHR_Integral CHR_LeftPar
CHR_Newline  CHR_Pi CHR_RightPar CHR_Sigma CHR_Space
CHR_UndScore CHR_[  CHR_]  CHR_{ CHR_}  CHR_<= CHR_>=
CHR_<>
```

```
$_R<<           ( $ "R\80\80" "R<angle><angle>" )
$_R<Z           ( $ "R\80Z"   "R<angle>Z"       )
$_XYZ           ( $ "XYZ"                       )
$_<<>>          ( $ "ABBB"                      )
$_{}            ( $ "{}"                        )
$_[]            ( $ "[]"                        )
$_''            ( $ "''"                        )
$_::            ( $ "::"                        )
$_LRParens      ( $ "()"                        )
$_2DQ           ( $ """"""                      )
$_ECHO          ( $ "ECHO"                      )
$_EXIT          ( $ "EXIT"                      )
$_Undefined     ( $ "Undefined"                 )
$_RAD           ( $ "RAD"                       )
$_GRAD          ( $ "GRAD"                      )
NEWLINE$        ( $ "\0a"                       )
SPACE$          ( $ " "                         )
```

6. Hex & Character Strings

6.1 Character Strings

The following words are avaliable for character string manipulation:

```
&$               ( $1 $2 --> $3 )
                   Appends $2 to $1
!append$         ( $1 $2 --> $3 )
                   Same as &$, except that it will attempt the concatenation
                   "in place," if there is not enough memory for the new
                   string, and the target is in tempob.
$>ID             ( $name --> Id )
                   Converts string object to name object
&$SWAP           ( ob $1 $2 --> $3 ob )
                   Appends $2 to $1, then swaps result with ob )
1-#1-SUB$        ( $ # --> $' )
                   Where $' = chars 1 thru #-1 of $
>H$              ( $ chr --> $' )
                   Prepends chr to $
>T$              ( $ chr --> $' )
                   Appends chr to $
AND$             ( $1 $2 --> $1 AND $2 )
                   Bitwise logical AND of two strings
APPEND_SPACE     ( $ --> $' )
                   Appends space to $
Blank$           ( # --> $  )
                   Creates a string of # spaces
CAR$             ( $ --> chr | $ )
                   Returns 1st chr of $ or NULL$ if $ is null
CDR$             ( $ --> $' )
                   $' is $ minus first character.  Returns NULL$ if $ is null
COERCE$22        ( $ --> $' )
                   If $ longer than 22 chars., truncates to 21 chars &
                   appends "..."
DECOMP$          ( ob --> $ )
                   Decompiles object for stack display
DO>STR           ( ob --> $ )
                   Internal version of ->STR
DROPNULL$        ( ob --> NULL$ )
                   Drops object, returns zero-length string
DUP$>ID          ( $name --> $name Id )
                   Dups, converts string object to name object
DUPLEN$          ( $ --> $ #length )
                   Returns $ and its length
DUPNULL$?        ( $ --> $ flag )
                   Returns TRUE if $ is zero-length
EDITDECOMP$      ( ob --> $ )
                   Decompile object for editing
JstGETTHEMESG    ( # --> $ )
                   Fetches message from message table
ID>$             ( ID --> $name )
                   Converts name object to a string
LAST$            ( $ # --> $' )
                   Returns last # chrs of $
```

```
LEN$            ( $ --> #length )
                  Returns length of $
NEWLINE$&$      ( $ --> $' )
                  Appends "\0a" to $
NULL$           ( --> $ )
                  Returns empty string
NULL$?          ( $ --> flag )
                  Returns TRUE if $ is zero-length
NULL$SWAP       ( ob --> $ ob )
                  Swaps empty string into level 2
NULL$TEMP       ( --> $ )
                  Creates empty string in TEMPOB
OR$             ( $1 $2 --> $3 )
                  Bitwise logical OR of two strings
OVERLEN$        ( $ ob --> $ ob #length )
                  Returns length of $ in level 2
POS$            ( $search $find #start --> #pos )
                  Returns #pos (#0 if not found) of $find
                  within $search starting at head of $search
POS$REV         ( $search $find #start --> #pos )
                  Returns #pos (#0 if not found) of $find
                  within $search starting at tail of $search
PromptIdUtil    ( id ob -> $ )
                  Returns string in the form "id: ob"
SEP$NL          ( $ --> $2 $1 )
                  Separate $ at newline character
SUB$            ( $ #start #end --> $' )
                  Returns substring of $
SUB$1#          ( $ #pos --> # )
                  Returns bint with value of character
                  in $ at position #pos
SUB$SWAP        ( ob $ #start #end --> $' ob )
                  Returns substring of $ and swaps with ob
SWAP&$          ( $1 $2 --> "$2$1" )
                  Appends $1 to $2
TIMESTR         ( %date %time --> "WED 03/30/90 11:30:15A" )
                  Returns string time and date
                  (like user word TSTR)
XOR$            ( $1 $2 --> $3 )
                  Bitwise logical XOR of two strings
a%>$            ( % --> $ )
                  Converts % to $ using current display mode
a%>$,           ( % --> $ )
                  Converts % to $ using current display mode
                  Same as a%>$, but with no commas
palparse        ( $ --> ob TRUE         )
                ( $ --> $ #pos $' FALSE )
                  Parse a string into an object and TRUE, or
                  returns position of error and FALSE
```

6.2 Hex Strings

```
#>%             ( hxs --> % )
                  Converts hxs to real
%>#             ( % --> hxs )
                  Converts real to hxs
&HXS            ( hxs1 hxs2 --> hxs3 )
                  Appends hxs2 to hxs1
2HXSLIST?       ( { hxs1 hxs2 } --> #1 #2 )
                  Converts list of two hxs into two bints
                  Generates Bad Argument Value error for
                  invalid input
HXS#HXS         ( hxs1 hxs2 --> %flag )
                  Returns %1 if hxs1 <> hxs2, otherwise %0
HXS>#           ( hxs --> # )
                  Converts lower 20 bits of hxs into a bint
HXS>$           ( hxs --> $ )
                  Does hxs>$, then appends base character
HXS>%           ( hxs --> % )
                  Converts hex string to real number
HXS<HXS         ( hxs1 hxs2 --> %flag )
                  Returns %1 if hxs1<hxs2, otherwise %0
HXS>HXS         ( hxs1 hxs2 --> %flag )
                  Returns %1 if hxs1>hxs2, otherwise %0
HXS>=HXS        ( hxs1 hxs2 --> %flag )
                  Returns %1 if hxs1>=hxs2, otherwise %0
HXS<=HXS        ( hxs1 hxs2 --> %flag )
                  Returns %1 if hxs1<=hxs2, otherwise %0
LENHXS          ( hxs --> #length )
                  Returns # of nibbles in hxs
NULLHXS         ( --> hxs )
                  Returns zero-length hex string
SUBHXS          ( hxs #m #n --> hxs' )
                  Returns substring
```

User RPL binary integers are actually hex strings. The following words assume 64-bit or shorter hex strings, and return results according to the current wordsize:

```
bit/            ( hxs1 hxs2 --> hxs3 )
Divides hxs1 by hxs2 bit%#/          ( % hxs --> hxs' )
                  Divides % by hxs, returns hxs
bit#%/          ( hxs % --> hxs' )                   Divides
hxs by %, returns hxs bit*           ( hxs1 hxs2 --> hxs3 )
                  Multiplies hxs1 by hxs2 bit%#*          (
% hxs --> hxs' )                  Multiplies % by hxs,
returns hxs bit#%*          ( hxs % --> hxs' )
                  Multiplies hxs by %, returns hxs
bit+            ( hxs1 hxs2 --> hxs3 )
Adds hxs1 to hxs2 bit%#+          ( % hxs --> hxs' )
                  Adds % to hxs, returns hxs
bit#%+          ( hxs % --> hxs' )                   Adds
hxs to %, returns hxs bit-           ( hxs1 hxs2 --> hxs3 )
                  Subtracts hxs2 from hxs1 bit%#-          (
% hxs --> hxs' )                  Suptracts % from hxs,
returns hxs bit#%-          ( hxs % --> hxs' )
                  Suptracts hxs from %, returns hxs
```

bitAND (hxs1 hxs2 --> hxs3)
Bitwise logical AND bitASR (hxs --> hxs')
Arithmetic shift right one bit
bitOR (hxs1 hxs2 --> hxs3)
Bitwise logical OR bitNOT (hxs1 hxs2 --> hxs3)
Bitwise logical NOT bitRL (hxs
--> hxs') Circular left shift by 1 bit
bitRLB (hxs --> hxs') Circular
left shift by 1 byte bitRR (hxs --> hxs')
Circular right shift by 1 bit
bitRRB (hxs --> hxs') Circular
right shift by 1 byte bitSL (hxs --> hxs')
Shift left by 1 bit bitSLB (hxs
--> hxs') Shift left by 1 byte
bitSR (hxs --> hxs') Shift
right by 1 bit bitSRB (hxs --> hxs')
Shift right by 1 byte bitXOR (
hxs1 hxs2 --> hxs3) Bitwise logical XOR

Wordsize control:

WORDSIZE (--> #) Returns user
binary integer wordsize dostws (# -->)
Stores binary wordsize hxs>$ (
hxs --> $) Converts hex string to chr
string using the current display mode and
wordsize

7. Real Numbers

Real numbers are written with %, and extended real numbers are written with %%.

7.1 Built-in Reals

The following real and extended real numbers are built in:

```
%%.1    %%4    %-8         %11    %21    %5
%%.4    %%5    %-9         %12    %22    %6
%%.5    %%60   %-MAXREAL   %13    %23    %7
%%0     %%7    %-MINREAL   %14    %24    %8
%%1     %-2    %.1         %15    %25    %MAXREAL
%%10    %-3    %.5         %16    %26    %MINREAL
%%12    %-4    %0          %17    %27    %PI
%%2     %-5    %1          %180   %3     %e
%%2PI   %-6    %10         %2     %360   %-1
%%3     %-7    %100        %20    %4
```

7.2 Real Number Functions

In the stack diagrams below, %1 and %2 refer to two different real numbers, NOT the real numbers one and two.

```
%%*              ( %%1 %%2 -->  %%3 )
                   Multiplies two extended reals
%%*ROT           ( ob1 ob2 %%1 %%2 --> ob2 %%3 ob1 )
                   Multiplies two extended reals,
                   then does a ROT
%%*SWAP          ( ob %%1 %%2 --> %%3 ob )
                   Multiplies two extended reals,
                   then does a SWAP
%%*UNROT         ( ob1 ob2 %%1 %%2 --> %%3 ob1 ob2 )
                   Multiplies two extended reals,
                   then does an UNROT
%%+              ( %%1 %%2 --> %%3 )
                   Adds two extended reals
%%-              ( %%1 %%2 --> %%3 )
                   Subtraction
%%ABS            ( %% --> %%' )
                   Absolute value
%%ACOSRAD        ( %% --> %%' )
                   Arc-cosine using radians
%%ANGLE          ( %%x %%y --> %%angle )
                   Angle using current angle mode from %%x %%y
%%ANGLEDEG       ( %%x %%y --> %%angle )
                   Angle using degrees from %%x %%y
```

```
%%ANGLERAD       ( %%x %%y --> %%angle )
                   Angle using radians from %%x %%y
%%ASINRAD        ( %% --> %%' )
                   Arc-sine using radians
%%CHS            ( %% --> %%' )
                   Change sign
%%COS            ( %% --> %%' )
                   Cosine
%%COSDEG         ( %% --> %%' )
                   Cosine using degrees
%%COSH           ( %% --> %%' )
                   Hyperbolic cosine
%%COSRAD         ( %% --> %%' )
                   Cosine using radians
%%EXP            ( %% --> %%' )
                   e^x
%%FLOOR          ( %% --> %%' )
                   Greatest integer <= x
%%H>HMS          ( %% --> %%' )
                   Decimal hours to hh.mmss
%%INT            ( %% --> %%' )
                   Integer part
%%LN             ( %% --> %%' )
                   ln(x)
%%LNP1           ( %% --> %%' )
                   ln(x+1)
%%MAX            ( %%1 %%2 --> %%3 )
                   Returns greater of two %%s
%%P>R            ( %%radius %%angle --> %%x %%y )
                   Polar to rectangular conversion
%%R>P            ( %%x %%y --> %%radius %%angle )
                   Rectangular to polar conversion
%%SIN            ( %% --> %%' )
                   Sine
%%SINDEG         ( %% --> %%' )
                   Sine using degrees
%%SINH           ( %% --> %%' )
                   Hyperbolic sine
%%SQRT           ( %% --> %%' )
                   Square root
%%TANRAD         ( %% --> %%' )
                   Tangent using radians
%%^              ( %%1 %%2 --> %%3 )
                   Exponential
%+               ( %1 %2 --> %3 )
                   Addition
%+SWAP           ( ob %1 %2 --> %3 ob )
                   Addition, then SWAP
%-               ( %1 %2 --> %3 )
                   Subtraction
%1+              ( % --> %+1 )
                   Adds one
%1-              ( % --> %-1 )
                   Subtracts one
```

```
%>#             ( % --> hxs )
                  Converts real to binary integer
%>%%            ( % --> %% )
                  Converts real to extended real
%>%%-           ( %1 %2 --> %%3 )
                  Converts 2 % to %%, then subtracts
%>%%1           ( %x --> %% )
                  Converts % to %%, then does 1/x
%>%%ANGLE       ( %x %y --> %%angle )
                  Angle in current angle mode
%>%%SQRT        ( % --> %% )
                  Converts % to %%, then sqrt(x)
%>%%SWAP        ( ob % --> %% ob )
                  Converts % to %%, then SWAP
%>C%            ( %real %imag --> C% )
                  Real to complex conversion
%>HMS           ( % --> %hh.mmss )
                  Decimal hours to hh.mmss
%ABS            ( % --> %' )
                  Absolute value
%ABSCOERCE      ( % --> # )
                  Absolute value, convert to bint
%ACOS           ( % --> %' )
                  Arc cosine
%ACOSH          ( % --> %' )
                  Hyperbolic arc cosine
%ALOG           ( % --> %' )
                  10^x
%ANGLE          ( %x %y --> %angle )
                  Angle using current angle mode from %x %y
%ASIN           ( % --> %' )
                  Arc sine
%ASINH          ( % --> %' )
                  Hyperbolic arc sine
%ATAN           ( % --> %' )
                  Arc tangent
%ATANH          ( % --> %' )
                  Hyperbolic arc tangent
%CEIL           ( % --> %' )
                  Next greatest integer
%CH             ( %1 %2 --> %3 )
                  Percent change
%CHS            ( % --> %' )
                  Change sign
%COMB           ( %m %n -> %COMB(m,n) )
                  Combinations of m items taken n at a time
%COS            ( % --> %' )
                  Cosine
%COSH           ( % --> %' )
                  Hyperbolic cosine
%D>R            ( % --> %' )
                  Degrees to radians
%EXP            ( % --> %' )
                  e^x
%EXPM1          ( $ --> %' )
                  e^x-1
```

```
%EXPONENT        ( % --> %' )
                   Returns exponent
%FACT            ( % --> %! )
                   Factorial
%FLOOR           ( % --> %' )
                   Greatest integer <= x
%FP              ( % --> %' )
                   Fractional part
%HMS+            ( %1 %2 --> %3 )
                   HH.MMSS addition
%HMS-            ( %1 %2 --> %3 )
                   HH.MMSS subtraction
%HMS>            ( % --> %' )
                   Convert hh.mmss to decimal hours
%IP              ( % --> %' )
                   Integer part
%IP>#            ( % --> # )
                   IP(ABS(x) converted to binary integer
%LN              ( % --> %' )
                   ln(x)
%LNP1            ( % --> %' )
                   ln(x+1)
%LOG             ( % --> %' )
                   Common log
%MANTISSA        ( % --> %' )
                   Returns mantissa
%MAX             ( %1 %2 --> % )
                   Returns larger of two reals
%MAXorder        ( %1 %2 --> %larger %smaller )
                   Orders two numbers
%MIN             ( %1 %2 --> % )
                   Returns smaller of two reals
%MOD             ( %1 %2 --> %3 )
                   Returns %1 MOD %2
%NFACT           ( % --> %' )
                   Factorial
%NROOT           ( %1 %2 --> %3 )
                   Nth root
%OF              ( %1 %2 --> %3 )
                   Returns percantage of %1 that is %2
%PERM            ( %m %n --> %PERM(%m,%n) )
                   Returns permutations of %m items
                   taken %n at a time
%POL>%REC        ( %x %y --> %radius %angle )
                   Rectangular to polar conversion
%R>D             ( %radians --> %degrees )
                   Radians to degrees
%RAN             ( --> %random )
                   Random number
%RANDOMIZE       ( %seed --> )
                   Updates random number seed, uses the
                   system clock if %=0
%REC>%POL        ( %radius %angle --> %x %y )
                   Polar to rectangular conversion
%SGN             ( % --> %' )
                   Sign: -1, 0 or 1 returned depending
                   on the sign of the argument
%SIN             ( % --> %' )
```

```
                  Sine
%SINH           ( % --> %' )
                  Hyperbolic sine
%SPH>%REC       ( %r %th %ph --> %x %y %z )
                  Spherical to rectangular conversion
%SQRT           ( % --> %' )
                  Square root
%T              ( %1 %2 --> %3 )
                  Percent total
%TAN            ( % --> %' )
                  Tangent
%TANH           ( % --> %' )
                  Hyperbolic tangent
%^              ( %1 %2 --> %3 )
                  Exponential
2%%>%           ( %%1 %%2 --> %1 %2 )
                  Extended real to real conversion
2%>%%           ( %1 %2 --> %%1 %%2 )
                  Real to extended real conversion
C%>%            ( C% --> %real %imag )
                  Complex to real conversion
DDAYS           ( %date1 %date2 --> %diff )
                  Days between dates in DMY format
DORANDOMIZE     ( % --> )
                  Updates random number seed
RNDXY           ( %number %places --> %number' )
                  Rounds %number to %places
TRCXY           ( %number %places --> %number' )
                  Truncates %number to %places
SWAP%>C%        ( %imag %real --> C% )
                  Real to complex conversion
```

8. Complex Numbers

Complex numbers are represented by C%, extended complex numbers by C%%.

8.1 Built-in Complex Numbers

```
C%0              (0,0)
C%1              (1,0)
C%-1             (-1,0)
C%%1             (%%1,%%0)
```

8.2 Conversion Words

```
%>C%             ( %real %imag --> C% )
%%>C%%           ( %%real %%imag --> C%% )
%%>C%            ( %%real %%imag --> C% )
C%>%             ( C% --> %real %imag )
C%%>%%           ( C%% --> %%real %%imag )
C%%>C%           ( C%% --> C% )
C%>%%            ( C% --> %%real %%imag )
C%>%%SWAP        ( C% --> %%imag %%real )
C>Im%            ( C% --> %imag )
C>Re%            ( C% --> %real )
```

8.3 Complex Functions

```
C%1/             ( C% --> C%' )
                   Inverse
C%ABS            ( C% --> % )
                   Returns SQRT(x^2+y^2) from (x,y)
C%ACOS           ( C% --> C%' )
                   Arc cosine
C%ALOG           ( C% --> C%' )
                   Common antilog
C%ARG            ( C% --> %)
                   Returns ANGLE(x,y) from (x,y)
C%ASIN           ( C% --> C%' )
                   Arc sine
C%ATAN           ( C% --> C%' )
                   Arc tangent
C%C^C            ( C%1 C%2 --> C%3 )
                   Power
C%CHS            ( C% --> C%' )
                   Change sign
C%%CHS           ( C%% --> C%%' )
                   Change sign
C%CONJ           ( C% --> C%' )
                   Conjugate
C%%CONJ          ( C%% --> C%%' )
                   Conjugate
```

```
C%COS           ( C% --> C%' )
                  Cosine
C%COSH          ( C% --> C%' )
                  Hyperbolic cosine
C%EXP           ( C% --> C%' )
                  e^z
C%LN            ( C% --> C%' )
                  Natural logarithm
C%LOG           ( C% --> C%' )
                  Common logarithm
C%SGN           ( C% --> C%' )
                  Returns (x/SQRT(x^2+y^2),y/SQRT(x^2+y^2)
C%SIN           ( C% --> C%' )
                  Sine
C%SINH          ( C% --> C%' )
                  Hyperbolic sine
C%SQRT          ( C% --> C%' )
                  Square root
C%TAN           ( C% --> C%' )
                  Tangent
C%TANH          ( C% --> C%' )
                  Hyperbolic tangent
```

9. Arrays

The notation [array] represents a real or complex array. [arry%] and [arryC%] represent real and complex arrays, respectively. {dims} means a list of array dimensions, which may be either { #cols } or { #rows #cols }.

Unless otherwise indicated, the following words do NOT check for out-of-range conditions (i.e. elements specified that are not within the range of the current array).

```
ARSIZE          ( [array] --> #elements )
                ( [array] --> {dims} )
GETATELN        ( # [array] --> ob TRUE )
                ( # [array] --> FALSE ) (no such element)
MAKEARRY        ( {dims} ob --> [array] )
                  Creates an unlinked array having the same
                  element type as ob.  All elements are
                  initialized to ob.
MATCON          ( [arry%] % --> [arry%]' )
                ( [arryC%] C% --> [arryC%]' )
                  Sets all elements in array to % or C%.
MATREDIM        ( [array] {dims} --> [array]' )
MATTRN          ( [array] --> [array]' )
MDIMS           ( [1-D array] --> #m FALSE )
                ( [2-D array] --> #m #n TRUE )
MDIMSDROP       ( [2-D array] --> #m #n )
                  Don't use MDIMSDROP on a vector!
OVERARSIZE      ( [array] ob --> [array] ob #elements )
PULLREALEL      ( [arry%] # --> [arry%] % )
PULLCMPEL       ( [arryC%] # --> [arryC%] C% )
PUTEL           ( [arry%] % # --> [arry%]' )
                ( [arryC%] C% # --> [arryC%] )
PUTREALEL       ( [arry%] % # --> [arry%]' )
PUTCMPEL        ( [arryC%] C% # --> [arryC%]' )
```

10. Composite Objects

The words described in this chapter are used for manipulating composite objects - mainly lists and secondaries. In the notation below, the term "comp" refers to either any composite object. The term "#n" refers to the number of objects in a composite object, and the term "#i" refers to the index of an object within a composite. The term "flag" refers to TRUE or FALSE.

```
&COMP           ( comp comp' --> comp'' ) comp is concatenated to comp'
2Ob>Seco        ( ob1 ob2 --> :: ob1 ob2 ; )
::N             ( obn ... ob1 #n --> :: obn ... ob1 ; )
::NEVAL         ( obn ... ob1 #n --> ? )
                  Does ::N, then evaluates secondary
>TCOMP          ( comp ob --> comp' ) ob is added to the tail of comp
CARCOMP         ( comp --> ob )
                ( comp --> comp )
                  Returns first object in the core of the
                  composite.  Returns an null comp if the
                  supplied composite is null.
CDRCOMP         ( comp --> comp' )
                ( comp --> comp )
                  Returns the core of the composite minus the
                  first object. Returns null comp if if the
                  supplied composite is null.
DUPINCOMP       ( comp --> comp obn ... ob1 #n )
DUPLENCOMP      ( comp --> comp #n )
DUPNULLCOMP?    ( comp --> comp flag ) TRUE if comp is null.
DUPNULL{}?      ( {list} --> {list} flag ) TRUE if {list} is null.
EQUALPOSCOMP    ( comp ob --> #pos | #0 )
                  Returns the index of the first object in comp
                  matching (EQUAL) ob (see NTHOF also)
Embedded?       ( ob1 ob2 --> flag )
                  Returns TRUE if ob2 is embedded in, or the
                  same as, ob1; otherwise returns FALSE.
INCOMPDROP      ( comp --> obn ... ob1 )
INNERCOMP       ( comp --> obn ... ob1 #n )
INNERDUP        ( comp --> obn ... ob1 #n #n )
LENCOMP         ( comp --> #n )
NEXTCOMPOB      ( comp #offset --> comp #offset' ob TRUE )
                ( comp #offset --> comp FALSE )
                  #offset is the nibble offset from the start
                  of the list to the Nth object in the list.
                  Returns a new #offset and the next object if
                  the next object is not SEMI, otherwise
                  returns the list and FALSE.  Use #5 at the
                  start of the list.
NTHCOMDDUP      ( comp #i --> ob ob )
NTHCOMPDROP     ( comp #i --> ob )
NTHELCOMP       ( comp #i --> ob TRUE )
                ( comp #i --> FALSE )
                  Returns FALSE if #i is out of range
NTHOF           ( ob comp --> #i | #0 ) Same as SWAP EQUALPOSCOMP.
NULL::          ( --> :: ; ) (Returns null secondary)
NULL{}          ( --> { } )  (Returns null list)
```

```
ONE{}N          ( ob --> { ob } )
Ob>Seco         ( ob --> :: ob ; )
POSCOMP         ( comp ob pred --> #i | #0 )
                  If the specified object "matches" an element
                  of the specified composite, where "match" is
                  defined as the specified predicate returning
                  TRUE when applied to an element of the comp
                  and the object, then POSCOMP returns the left-
                  to- right index of the element within the
                  composite, or zero.  For instance, to find the
                  first real less than 5 in a list of reals:

                            :: {list} 5 ' %< POSCOMP ;

PUTLIST         ( ob #i {list} --> {list}' ) (Assumes 0<#i<=#n)
SUBCOMP         ( comp #m #n --> comp' ) (Returns subcomposite)
                  IF #m > #n THEN comp' is null
                  IF #m=0    THEN #m is set to 1
                  IF #n=0    THEN #n is set to 1
                  IF #m > LEN(comp) THEN comp' is null
                  IF #n > LEN(comp) THEN #n is set to LEN(comp)
SWAPINCOMP      ( comp obj --> obj obn ... ob1 #n )
THREE{}N        ( ob1 ob2 ob3 --> { ob1 ob2 ob3 } )
TWO{}N          ( ob1 ob2 --> { ob1 ob2 } )
{}N             ( obn ... ob1 #n --> {list} )
apndvarlst      ( {list} ob --> {list}' )
                  Adds ob to the list if ob is not found within
                  the list
matchob?        ( ob comp --> ob TRUE )
                ( ob comp --> FALSE )
                  Determines if ob is equal (EQUAL) to any element of comp
```

11. Tagged Objects

The following words are available for manipulating tagged objects. Remember that an object can have multiple tags.

```
%>TAG           ( ob % --> tagged )
                  Tags ob with %

>TAG            ( ob $ --> tagged )
                  Tags ob with $

ID>TAG          ( ob id/lam --> tagged )
                  Tags ob with id

STRIPTAGS       ( tagged --> ob )
                  Removes all tags

STRIPTAGSl2     ( tagged ob' --> ob ob' )
                  Strips tags from level 2 object

TAGOBS          ( ob $ --> tagged )
                ( ob1 ... obn { $1 ... $n }
                  --> tagged1 ... taggedn )
                  Tags one object, or several objects
                  if a list of tags is in level 1

USER$>TAG       ( ob $ --> tagged )
                  Tags ob with $ (up to 255 chrs valid)
```

12. Unit Objects

When unit objects are compared for dimensional consistency, a hex string, called a "quantity string", may be extracted using the word U>NCQ. This quantity string contains information about which units are contained, and can be directly compared with another quantity string. If the quantity strings match, the two unit objects can be said to be dimensionally consistent. U>NCQ also returns extended real numbers consisting of the number and a conversion factor to base units.

```
U>NCQ           ( unit --> n%% cf%% qhxs )
                  Returns number, conversion factor,
                  and hex quantity string
UM=?            ( unit1 unit2 --> %flag )
                  Returns %1 if two unit obs are equal
UM#?            ( unit1 unit2 --> %flag )
                  Returns %1 if unit1 <> unit2
UM<?            ( unit1 unit2 --> %flag )
                  Returns %1 if unit1 < unit2
UM>?            ( unit1 unit2 --> %flag )
                  Returns %1 if unit1 > unit2
UM<=?           ( unit1 unit2 --> %flag )
                  Returns %1 if unit1 <= unit2
UM>=?           ( unit1 unit2 --> %flag )
                  Returns %1 if unit1 >= unit2
UM>U            ( % unit --> unit' )
                  Replaces the number part of a unit object
UM%             ( unit %percentage --> unit' )
                  Returns a percentage of a unit object
UM%CH           ( unit1 unit2 --> % )
                  Returns percent difference
UM%T            ( unit1 unit2 --> % )
                  Returns percentage fraction
UM+             ( unit1 unit2 --> unit3 )
                  Addition
UM-             ( unit1 unit2 --> unit3 )
                  Subtraction
UM*             ( unit1 unit2 --> unit3 )
                  Multiply
UM/             ( unit1 unit2 --> unit3 )
                  Divide
UM^             ( unit1 unit2 --> unit3 )
                  Power
UM1/            ( unit --> unit' )
                  Inverse
UMABS           ( unit --> unit' )
                  Absolute value
UMCHS           ( unit --> unit' )
                  Change sign
UMCONV          ( unit1 unit2 --> unit1' )
                  Converts unit1 to units of unit2
UMCOS           ( unit --> unit' )
                  Cosine
UMMAX           ( unit1 unit2 --> unit? )
```

```
                  Returns larger of unit1 and unit2
UMMIN           ( unit1 unit2 --> unit? )
                  Returns smaller of unit1 and unit2
UMSI            ( unit --> unit' )
                  Convert to SI base units
UMSIN           ( unit --> unit' )
                  Sine
UMSQ            ( unit --> unit' )
                  Square
UMSQRT          ( unit --> unit' )
                  Square root
UMTAN           ( unit --> unit' )
                  Tangent
UMU>            ( unit --> % unit' )
                  Returns number and normalized unit parts
                  of a unit object
UMXROOT         ( unit1 unit2 --> unit3 )
                  unit1^1/unit2
UNIT>$          ( unit --> $ )
                  Decompiles a unit object with tics
```

13. Temporary Variables and Temporary Environments

One of the features implemented in RPL is the capability of creating temporary variables (aka "local variables", "lambda variables") whose names are given by the programmer, and which can be destroyed easily when they are no longer needed. These temporary variables serve a number of important purposes. First of all, they can be used to eliminate stack manipulations within a program, which makes the task of keeping track of the stack much easier, and makes debugging easier. In addition, they are essential for the implementation of programs which take an indefinite number of parameters and want to save one or more of those parameters.

Temporary variables are referenced by temporary identifier objects ("local names"), and the binding between a temporary identifier object and its value is supported by structures in memory called temporary environments. (This is the RPL analogue of LISP "lambda binding").

Temporary environments are stacked in chronological order. This allows the programmer the opportunity to create his own "private" temporary variables, without the possibility of interfering with those created by others. When a temporary identifier object is executed, a search is made through the stack of temporary environments, starting in the most recently created and working back through previous environments if necessary. When a match is made between the temporary identifier object being executed and a temporary identifier object in one of the temporary environments, the object bound to that identifier is pushed onto the data stack. Executing an unbound temporary identifier object is an error condition.

The processes of creating a temporary environment and assigning initial values to its temporary variables are accomplished simultaneously with the provided object BIND. BIND expects a list of temporary identifier objects on the top of the data stack and at least as many objects (excluding the list itself) on the stack as there are temporary identifier objects in the list. BIND will then create a temporary environment and bind each temporary identifier object in the list with an object on the stack, removing that object from the stack.

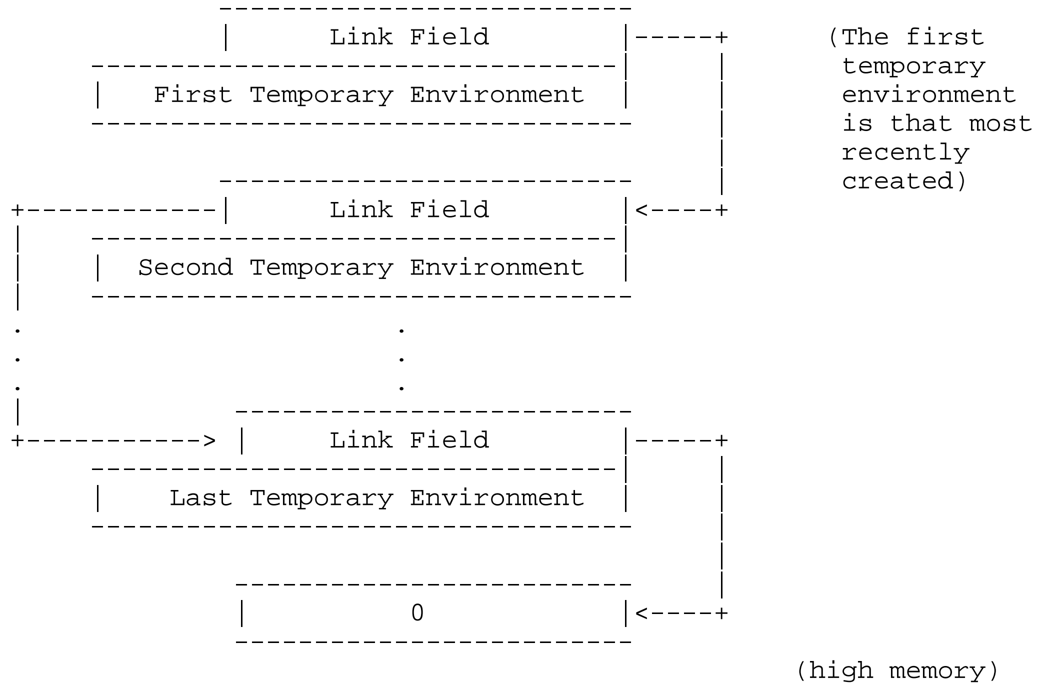
Subsequent execution of any of the temporary identifier objects in the list will return the object bound to it. The value bound to a temporary identifier object can be changed using STO in exactly the same manner as a value "bound" to an identifier object (global name).

The dissolution of a temporary environment is accomplished with the provided object ABND (short for "abanbon"). ABND removes the top-most temporary environment from the stack of temporary environments. Individual temporary variables cannot be removed from a temporary environment; the temporary environment as a whole must be abandoned.

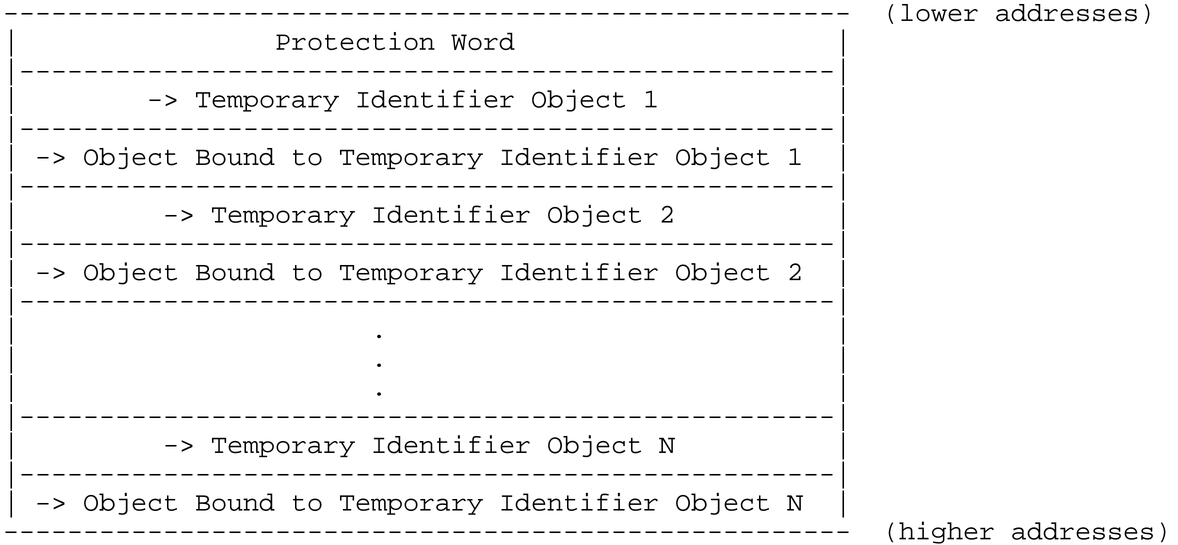
Note that the RPL compiler does not check to see if there is an ABND to match each BIND. You can include the two within a single program, or put them in separate programs as you like with no restrictions other than the requirements of good structured programming practice. This also means that you must remember to include the ABND at some point, otherwise you may leave unnecessary environments around after a program has completed execution. (In user RPL, you do not have such freedom. The structure word -> has BIND built into it, and the command line parser demands that there be a matching >> or ' that includes ABND.)

13.1 Structure of the Temporary Environment Area

The structure of the temporary environment area is shown below.



Each temporary environment consists of a protection word (a binary integer object body) which is used in error handling, followed by a sequence of one or more pairs of object pointers. The first object pointer in each pair is the address of a temporary identifier object and the second object pointer in each pair is the address of the object bound to that temporary identifier object. All of the object pointers in a temporary environment are updatable. The structure of each temporary environment within the temporary environment area is shown below.



13.2 Named vs. Unnamed Temporary Variables

Temporary variables are normally named by the corresponding temporary identifier in the list used by BIND. The names in the list are used in the same order as the bound objects appear on the stack--the last identifier in the list corresponds to the object in level 1, the next-to-last identifier corresponds to the object in level 2, and so on. In the following example, the binary integer ONE is bound into Var1, and TWO is bound into Var2:

```
    ONE TWO
    {
      ' LAM Var1
      ' LAM Var2
    }
    BIND                ( Binds ONE into temporary variable Var1, and
                           TWO into variable Var2 )
    ...
    LAM Var1            ( Recalls ONE from Var1 )
    ...
    LAM Var2            ( Recalls TWO from Var2 )
    ...
    ' LAM Var1 STO      ( Stores new object in Var1 )
    ...
    ABND                ( Abandons temp env. )
```

Temporary identifiers may contain any text characters, except that you should not start the names with ' or # as such names are reserved for the built-in ROM programs. For similar reasons, it is recommended that you use names that can not conflict with user-generated names; an easy way to insure this is to include an "illegal" character such as one of the object delimiters in your names.

If there is NO CHANCE that another temporary environment will be created above the environment you are about to create, null names may be used to save memory. There are a number of utility words that allow you to access local variables in the topmost environment by position number, which is faster than the ordinary name resolution. For example, the example above would look like this:

```
::
  ONE TWO
  { NULLLAM NULLLAM }
  BIND               ( Binds ONE and TWO into nullnamed temporary
                       variables )
  ...
  2GETLAM            ( Recalls ONE from first variable )
  ...
  1GETLAM            ( Recalls TWO from last variable )
  ...
  2PUTLAM            ( Stores new object in first variable )
  ...
  ABND               ( Abandons temp environment. )
;
```

The numbering starts with the last temporary variable (i.e. in the same order as the stack level number).

13.3 Provided Words for Temporary Variables

The following words are provided for working with temporary variables. The term "lamob" is used in this case to indicate an object recalled from a termporary variable.

```
1ABNDSWAP       ( ob --> lamob ob )
                  Does :: 1GETLAM ABND SWAP ;
1GETABND        ( --> lamob )
                  Does :: 1GETLAM ABND ;
1GETLAM
 ...            ( --> ob )
22GETLAM         Returns contents of Nth lam

1GETSWAP        ( ob --> lamob ob )
                  Does :: 1GETLAM SWAP ;
1LAMBIND        ( ob --> )
                  Does :: 1NULLLAM{} BIND ;
1NULLLAM{}      ( --> { NULLLAM } )
                  Returns list with one null lam
1PUTLAM
 ...            ( ob --> )
22PUTLAM        ( Stores ob into Nth lam

2GETEVAL        ( --> ? )
                  Recalls & evaluates ob in 2nd lam
@LAM            ( id --> ob TRUE )
                ( id --> FALSE )
                  Recalls lam by name, returns ob and
                  TRUE if id exists; FALSE otherwise
ABND            ( --> )
                  Abandons topmost temp var env.
BIND            ( ob ... { id ... } --> )
                  Creates new temp var env.
CACHE           ( obn ... ob1 n lam --> ) Saves away n objects plus the count
                  n in a temporary environment, each object being bound to
the
                  same identifier lam.  The last pair has the count. )
DUMP            ( NULLLAM --> ob1..obn n ) DUMP is essentially the inverse of
                  CACHE, BUT: it ONLY works with NULLLAM as the cached name,
                  and it ALWAYS does a garbage collect.
DUP1LAMBIND     ( ob --> ob )
                  Does DUP, then 1LAMBIND
DUP4PUTLAM      ( ob --> ob )
                  Does DUP, then 4PUTLAM
DUPTEMPENV      ( --> )
                  Duplicates topmost temporary env.,
                  clearing the protection word.
GETLAM          ( #n --> ob )
                  Returns object in #nth temp var
NULLLAM         ( --> NULLLAM )
                  Null temporary variable name
PUTLAM          ( ob #n --> )
                  Stores ob in #nth temp var
STO             ( ob id --> )
                  Stores ob in named global/temp var
STOLAM          ( ob id --> )
                  Stores ob in named temp var
```

13.4 Coding Suggestions

The DEFINE feature of the RPL compiler can be used to combine the legibility of named variables with the speed and efficiency of null-named variables. For example:

```
  DEFINE RclCode  1GETLAM
  DEFINE StoCode  1PUTLAM
  DEFINE RclName  2GETLAM
  DEFINE StoName  2PUTLAM
  ::
    ...
    { NULLLAM NULLLAM }
    BIND                ( Binds two objects into nullnamed
                          temp variables 1 and 2 )
    ...
    RclCode             ( Recalls contents of last variable )
    ...
    RclName             ( Recalls contents of first variable )
    ...
    StoCode             ( Stores object in first variable )
    ...
    ABND                ( Abandons temp environment. )
  ;
```

If a large number of temporary variables are to be used without names, here is a code-saving tip:

Replace:

```
        ...
        {
          NULLLAM NULLLAM NULLLAM NULLLAM
          NULLLAM NULLLAM NULLLAM NULLLAM
          NULLLAM NULLLAM NULLLAM NULLLAM
          NULLLAM NULLLAM NULLLAM NULLLAM
          NULLLAM NULLLAM NULLLAM NULLLAM
          NULLLAM NULLLAM NULLLAM NULLLAM
        } BIND
        ...
```

With:

```
        NULLLAM TWENTYFOUR NDUPN
        {}N BIND
```

The first method takes 67.5 bytes, whereas the latter method takes 12.5 bytes, so there's a savings of 55 bytes!

You can also use TWENTYFOUR ' NULLLAM CACHE, which is shorter yet and does not require building the list of null identifiers in tempob. Note, however, that CACHE adds an extra temporary variable (to hold the count), so all of the variable position numbers differ by one from the previous methods.

14. Checking Arguments

Any program object which can be executed directly by a user should insure that the correct number and types of arguments are present to prevent problems. If the object is ultimately to be a library command, then it should follow the command structure convention (see section xxx):

```
        :: CK0 ... ; for 0 argument commands, or

        :: CK<n>&Dispatch type1 action1
                          type2 action2
                               ...
                          typen actionn
        ;
```

for <n> argument commands, where typei is a type code and
actioni is the corresponding dispatchee for that type combination, or

```
        :: CKN ... ;  for commands that take an number of
```

arguments specified
by a real number in level 1 (like PICK or ->LIST).

CK<n>&Dispatch is actually a combination of CK<n> and CK&DISPATCH1. There are a few built-in commands (e.g. TYPE) that use the two words instead of the combined form, but all algebraic functions must use CK<n>&Dispatch since these words also serve to identify the argument count used by a function.

If an object is not intended as a library command, then it should have the following structure:

```
        :: CK0NOLASTWD ... ; for 0 argument programs, or

        :: CK<n>NOLASTWD CK<n>&DISPATCH1 type1 action1
                          type2 action2
                               ...
                          typen actionn
        ;
```

for <n> argument programs, or

```
        :: CKNNOLASTWD ... ; for programs that take
```

arguments as specified
in level 1.

14.1 Number of Arguments

The following words verify that from 0-5 arguments are on the stack, and issue the "Too Few Arguments" error otherwise.

CK0, CK0NOLASTWD	No arguments required
CK1, CK1NOLASTWD	One argument required
CK2, CK2NOLASTWD	Two arguments required
CK3, CK3NOLASTWD	Three arguments required
CK4, CK4NOLASTWD	Four arguments required
CK5, CK5NOLASTWD	Five arguments required

Each word CK<n>... "marks" the stack below the <n>th argument, and if argument recovery is in effect, saves a copy of the <n> arguments in the last argument save area. If an error occurs that is handled by the outer loop error handler, then the stack is cleared to the marked level (this removes any stray objects that were not put there by the user). If the argument recovery system is active, then the saved arguments are restored to the stack.

Any CK<n> also records the command in which it is executed, again for the sake of the outer loop error handler, which uses the command name as part of the error message display. A CK<n> should only be used in library commands, and must be the first object in the command program. CK<n>NOLASTWD does not record the command, and may be used at any point. However, it generally not a good idea to execute these words except

* at the beginning of a user-executed object, or

* immediately after the execution of any user procedure.

User procedures should only be executed when the stack contains only user objects; the CK<n>NOLASTWD (usually CK0NOLASTWD) is executed immediately after the user procedure to update the stack save mark to protect the stack results of the procedure. This is usually done in conjunction with 0LASTOWDOB!, which clears the command save done by the last CK<n> executed within the user procedure, so that that command is not identified as the culprit for any subsequent errors. Useful words for these purposes are

```
        AtUserStack     which is :: CK0NOLASTWD 0LASTOWDOB! ;
        CK1NoBlame      which is :: 0LASTOWDOB! CK1NOLASTWD ;
```

For objects that take a stack specified number of arguments, the analogs to CK<n> and CK<n>NOLASTWD are CKN and CKNNOLASTWD. Both words check for a real number in level 1, then check if there are that many additional objects on the stack. The stack is marked at level 2, and only the real number is restore by LAST ARG.

14.2 Dispatching on Argument Type

The words CK&DISPATCH1 and CK&DISPATCH0 provide a dispatch-by-type mechanism (the CK<n>&Dispatch words include the same mechanism, so the following discussion applies to them as well), that provides straightforward branching according to the object types of up to five arguments at a time. Each word is followed by an indefinite number of pairs of object. Each pair consists of a binary integer or object pointer to a binary integer, followed by any object or object pointer (exclusive use of object pointers guarantees the fastest dispatching):

```
  ...
  CK&DISPATCH1   #type1 action1                  #type2
action2                 ...                     #typen
action3
  ;
```

The object-pair sequence must be terminated by a SEMI (;).

CK&DISPATCH1 proceeds as follows: For each typei, from type1 to typen, if typei matches the stack configuration then execute actioni, discarding the rest of word containing CK&DISPATCH1. If no match is found, report the error "Bad Argument Type".

If a complete pass is made through the table without a successful match, the CK&DISPATCH1 makes a second pass through the table, this time stripping any tags from stack objects and matching the remaining objects against the required types.

The word CK&DISPATCH0 does not perform the second pass which strips tags. This word should only be used where it is important to find a tagged object. The general behavior of the HP 48 is to regard tags as being auxiliary to the tagee, and thus CK&DISPATCH1 should be used in most cases.

A binary integer typei is nominally encoded as follows:

```
#nnnnn
 |||||
 ||||+-- Level 1 argument type
 |||+--- Level 2 argument type
 ||+---- Level 3 argument type
 |+----- Level 4 argument type
 +------ Level 5 argument type
```

Each "n" is a hexadecimal digit representing an object type, as shown in the table below. Thus #00011 represents two real numbers; #000A0 indicates a symbolic class object (symb, id, or lam) in level 2 and any type of object in level 1. There are also two-digit object type numbers, ending in F; use of any of these consequently reduces the total number of arguments that can be encoded in a single typei integer. For example, #13F4F represents a real number in level 3, an extended real in level 2, and an extended complex in level 1.

The following table shows the hex digit values for each argument type. The column "# name" shows the object pointer name for the corresponding binary integer that may be used for a single argument function. The "Binary Integers" chapter contains a list of built-in binary integers that may be used for various common two-argument combinations.

Value	Argument	# name	User TYPE
0	Any Object	any	
1	Real Number	real	0
2	Complex Number	cmp	1
3	Character String	str	2
4	Array	arry	3,4
5	List	list	5
6	Global Name	idnt	6
7	Local Name	lam	7
8	Secondary	seco	8
9	Symbolic	symb	9
A	Symbolic Class	sym	6,7,9
B	Hex String	hxs	10
C	Graphics Object	grob	11
D	Tagged Object	TAGGED	12
E	Unit Object	unitob	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	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

14.3 Examples

Built-in commands and other words provide good examples of the check-and- dispatching scheme. The following is the definition of the user command STO:

```
:: CK2&Dispatch
    THIRTEEN  XEQXSTO                    ( 2:any object 1:tagged object)
    SIX       :: STRIPTAGSl2 ?STO_HERE ; ( 2:any        1:id )
    SEVEN     :: STRIPTAGSl2 STO ;       ( 2:any        1:lam )
    NINE      :: STRIPTAGSl2 SYMSTO ;    ( 2:any        1:symb )
    # 000c8   PICTSTO                    ( 2:grob       1:program [PICT] )
    # 009f1   LBSTO                      ( 2:backup ob  1:real number )
    # 008f1   LBSTO                      ( 2:library    1:real number )
;
```

Since STO is a command, it starts with CK2&Dispatch, which verifies that there are two arguments present, saves those arguments and the command STO for error handling, then dispatches to one of the action objects listed in the dispatch table. If the level one object is tagged, STO dispatches to the word XEQSTO. For a global name (id), STO executes :: STRIPTAGSl2 ?STO_HERE ;, which is directly embedded in the STO program. And so forth, down to the last choice, which is a dispatch to LBSTO when the arguments are a library in level 2, and a real number in level 1.

The TYPE command provides an example of dispatching at a point other than the start of a command. TYPE is a command, but its argument counting and argument type dispatching are separated so that the latter part can be called by other system words that don't want to mark the stack:

```
::
  CK1
  :: CK&DISPATCH0
        real             %0
        cmp              %1
        str              %2
        arry             XEQTYPEARRY
        list             %5
        id               %6
        lam              %7
        seco             TYPESEC ( 8, 18, or 19 )
        symb             %9
        hxs              %10
        grob             % 11
        TAGGED           % 12
        unitob           % 13
        rompointer       % 14
        THIRTYONE ( # )  % 20
        rrp              % 15
        # 3F ( %% )      % 21
        # 4F ( C%% )     % 22
        # 5F ( LNKARRY ) % 23
        # 6F ( CHR )     % 24
        # 7F ( CODE )    % 25
        library          % 16
```

```
        backup            % 17
        # AF              % 26 ( Library Data )
        any               % 27 ( external )
  ;
  SWAPDROP
;
```

CK&DISPATCH0 is used here, although CK&DISPATCH1 would work as well since tagged objects are explicitly listed in the dispatch table. Notice also that the last typei is "any", meaning that type 27 is returned for any object type not previously listed.

The "inner" program (starting after the CK1) is the body of the system word XEQTYPE.

15. Loop Control Structures

Two types of looping structures are available - indefinite loops and definite loops.

15.1 Indefinite Loops

Indefinite loops are constructed from combinations of the following RPL words:

BEGIN (-->)
Copies the interpreter pointer (RPL variable I) onto the return stack. Also called IDUP.

UNTIL (flag -->)
If flag is TRUE, drops the top pointer on the return stack, otherwise copies that pointer to the interpreter pointer.

WHILE (flag -->)
If the flag is TRUE, then does nothing. Else drops the first pointer from the return stack, and skips the interpreter pointer past the next two objects.

REPEAT (-->)
-->
Copies the first pointer on the return stack to the interpreter pointer.

AGAIN (-->)

The WHILE loop is an indefinite loop:

```
BEGIN
  <test clause>
WHILE
  <loop object>
REPEAT
```

The WHILE loop executes <test clause>, and if the result is the system flag TRUE, executes the <loop object> and repeats; otherwise it exits to past the REPEAT. The WHILE loop never executes if the first run of <test clause> returns FALSE.

The action of WHILE requires <loop object> to be a single object. However, the RPL compiler automatically combines multiple objects between WHILE and REPEAT into a program object, so that

```
BEGIN
  <test clause>
WHILE
  ob1 ... obn
REPEAT
```

is actually compiled as

```
BEGIN
  <test clause>
WHILE
  :: ob1 ... obn ;
REPEAT
```

Another common indefinite loop is the BEGIN...UNTIL:

```
BEGIN
  <loop clause>
UNTIL
```

This loop executes at least once, as opposed to the WHILE loop, which does not execute its loop object if the initial test is false. The word UNTIL expects a flag (TRUE or FALSE).

The BEGIN...AGAIN loop has no test:

```
BEGIN
  <loop clause>
AGAIN
```

Terminating this loop requires an error event, or a direct manipulation of the return stack.

15.2 Definite Loops

Definite loops with a loop counter are achieved in RPL by means of the DO Loop. The word DO takes two binary integer objects from the stack, and stores the top object as the index and the other as the stopping value in a special DoLoop environment. DO also copies the interpreter pointer onto the return stack. DoLoop environments are stacked, so that they can be nested indefinitely. The topmost index is recalled by INDEX@; the index in the second environment by JINDEX@. The topmost stopping value is available via ISTOP@.

DO's counterparts are LOOP and +LOOP. LOOP increments the index value in the topmost DoLoop environment; then, if the (new) value is greater than or equal to the stopping value, LOOP drops the top pointer from the return stack and removes the topmost DoLoop environment. Otherwise, LOOP acts copies the top return stack pointer to the interpreter pointer. The standard form of a DoLoop is

```
stop start DO <loop clause> LOOP,
```

which executes <loop clause> for each value of an index from start to stop-1.

+LOOP is similar to LOOP, except that it takes a binary integer from the stack and increments the loop counter by that amount rather than 1.

15.2.1 Provided_Words

The following words are provided for use with DO loops. Words marked with * are not recognized as special by the RPL compiler, so you should include compiler directives to prevent warning messages. For example, #1+_ONE_DO can be followed by (DO) which matches the following LOOP for the sake of the compiler but does not generate any compiled code.

```
#1+_ONE_DO *     ( #finish --> )
                   Equivalent to #1+ ONE DO; commonly used to execute a loop
                   #finish times.
DO               ( #finish #start --> )
                   Begins DO loop
DROPLOOP *       ( ob --> )
                   Performs DROP, then LOOP
DUP#0_DO *       ( # --> # )
                   Begins # ... #0 DO loop
DUPINDEX@        ( ob --> ob ob #index )
                   Does DUP, then returns value of index in topmost DoLoop
env.
ExitAtLOOP       ( --> )
                   Stores zero in stopping value of topmost DoLoop environment
INDEX@           ( --> #index )
                   Returns index of topmost DoLoop environment
INDEX@#-         ( # --> #' )
                   Subtracts index value of topmost
```

```
                  DoLoop environment from #
INDEXSTO        ( # --> )
                  Stores # as index of top DoLoop environment
ISTOP@          ( --> #stop )
                  Returns stop value of the topmost DoLoop environment
ISTOPSTO        ( # --> )
                  Stores new stop value in the topmost DoLoop environment
JINDEX@         ( --> #index )
                  Returns index of second DoLoop environment
LOOP            ( --> )
                  End of loop structure
NOT_UNTIL *     ( flag --> )
                  End of loop structure
ONE_DO *        ( #finish --> )
                  Begins #1...#finish DO loop
OVERINDEX@      ( ob1 ob2 --> ob1 ob2 ob1 #index )
                  Does OVER, then returns value of
                  index in topmost DoLoop environment
SWAPINDEX@      ( ob1 ob2 --> ob2 ob1 #index )
                  Does SWAP, then returns value of index in topmost
                  DoLoop environment
SWAPLOOP *      ( ob1 ob2 --> ob2 ob1 )
                  Does SWAP, then LOOP
ZEROISTOPSTO    ( --> )
                  Stores zero as the stop value in the topmost DoLoop
                  environment
ZERO_DO *       ( #finish --> )
                  Begins DO loop from #0 to #finish
toLEN_DO        ( {list} --> {list} )
                  Begins DO loop from #1 of elements in list to stop value
                  #number-of-elements+1.
```

15.2.2 Examples

```
FIVE ZERO
DO
  INDEX@
LOOP
```

This returns the values:

```
#00000 #00001 #00002 #00003 #00004
```

The following sequence displays each of the elements (up to 8) of a list of strings on a separate display line.

```
DUPLENCOMP
ONE_DO (DO)
  DUP INDEX@ NTHCOMPDROP
  INDEX@ DISPN
LOOP
```

A more compact version uses toLEN_DO:

```
toLEN_DO (DO)
  DUP INDEX@ NTHCOMPDROP
  INDEX@ DISPN
LOOP
```

Another version is slightly faster, since it avoids repeated extraction of list elements:

```
INNERCOMP
#1+_ONE_DO (DO)
  INDEX@ DISPN
LOOP
```

This version displays the elements in reverse order relative to the previous versions.

16. Error Generation & Trapping

The RPL error handling sub-system is invoked by execution of the word ERRJMP, that is, when a procedure class object wishes to generate an error, it executes ERRJMP (probably after setting the values of ERROR and ERRNAME). The mechanics of ERRJMP will be described later.

16.1 Trapping: ERRSET and ERRTRAP

RPL provides procedure objects with the capability to intercept execution of the error handling sub-system, that is, trap an error generated by an object which is lower on the threaded order. This capability is made available via the built-in objects ERRSET and ERRTRAP used in the following way:

```
:: ... ERRSET <suspect object> ERRTRAP <if-error object> ... ;
```

In the above, an error generated by <suspect object> is to be trapped. <if-error object> denotes the object to be executed if <suspect object> generates an error. The exact algorithm is: If <suspect object> generates an error, then continue execution at <if-error object>; else, continue execution beyond <if-error object>.

The action of <if-error object> is completely flexible; when <if-error object> gets control, it may examine the values of ERROR and ERRNAME to determine whether or not it is even concerned with the current error. If not, it may simply re-start the sub-system by executing ERRJMP. If so, it may decide to handle the error, that is, clear both ERROR and ERRNAME and NOT restart the sub-system. It may also disable execution of the remainder of the program (perhaps via RDROP).

Note that throughout (normal) execution of <suspect object>, an object pointer to the following ERRTRAP is somewhere in the runstream.

16.2 Action of ERRJMP

When an RPL procedure wants to initiate an error, it executes ERRJMP, which the error handling sub-system. ERRJMP cycles through the RUNSTREAM from the interpreter pointer I up through the return stack searching for an error trap. Specifically, ERRJMP removes pending program bodies from the RUNSTREAM until it finds one whose first element is an object pointer addressing ERRTRAP (this program body may correspond to a return stack level as well as the interpreter pointer I). It then SKIPs over the object pointer to ERRTRAP and continues execution beyond it (at the <if-error object>).

Note, therefore, that ERRTRAP is only executed if <suspect object> terminates without generating an error; in this

case, ERRTRAP will, among other things, SKIP <if-error object> and continue execution beyond it.

If a procedure is not merely passing along an error that it did not initiate, its invokation of ERRJMP should be preceded by execution of ERRORSTO, which stores an error number in a special system location. ERROR@ returns the stored error number, which error traps can use to determine if they want to handle a particular error. The error number is stored and returned as a binary integer; the high-order 12 bits of the number represent the Library ID of the library containing the error message, and the remaining bits indicate the error number within the library's message table.

16.3 The Protection Word

Each temporary environment and each DoLoop environment has a protection word. The sole reason for the existence of this protection word is to allow the error handling sub-system to distinguish temporary and DoLoop environments that were in existence at the time an error trap was set from those which came into being after the error trap was set. For example, consider the following:

```
::
  ...
  { NULLLAM } BIND
  ...
  TEN ZERO DO
    ERRSET ::
      ...
      { NULLLAM } BIND
      ...
      FIVE TWO DO
        <procedure>
      LOOP
      ABND
    ;
    ERRTRAP
      :: "Procedure Failed" FlashMsg ;
  LOOP
  ...
  ABND
  ...
;
```

If <procedure> generates an error, then this error will be trapped by the word or secondary following ERRTRAP. However, the inner DoLoop and temporary environments must be deleted so that the outer procedure has available the correct DoLoop parameters and local variables. The protection word serves to abet this function.

ERRSET increments the protection word in the topmost

temporary environment and the topmost DoLoop environment. These topmost environments therefore have a non-zero protection word. (DO and BIND always initialize the protection word to zero).

ERRTRAP and ERRJMP delete temporary and DoLoop environments (from the first to the last) until, in both cases, they find one with a non-zero protection word, which is then decremented. Therefore, whenever either ERRJMP executes at <if-error object> or ERRTRAP executes past <if-error object>, only temporary and DoLoop environments which existed at the ERRSET will be present.

Note especially that the protection word is more than just a switch so as to allow a practically indeterminant level of nesting of error traps.

The example above is actually a poorly formed error trap - the code should actually determine what the error was, and take action accordingly. The word ERROR@ may be used to recall which error occurred. The error numbers correspond to the message numbers - see the message table in appendix A of the "HP48 Programmers Reference Manual".

16.4 Error Words

The following words are provided for error management:

```
ABORT           ( --> )
                  Does ERRORCLR and ERRJMP
DO#EXIT         ( msg# --> )
                  Stores a new error number and executes ERRJMP;
                  also executes AtUserStack
                  Puts the object ERRJMP on the stack
ERRBEEP         ( --> )
                  Generates an error beep
ERRJMP          ( --> )
                  Invokes error handling subsystem
ERROR@          ( --> # )
                  Returns the current error number
ERRORCLR        ( --> )
                  Stores zero as the error number
ERROROUT        ( # --> )
                  Stores a new error number and does ERRJMP
ERRORSTO        ( # --> )
                  Stores new error number
ERRTRAP         ( --> )
                  Skips next object in runstream.
```

17. Test and Control

This chapter reviews words related to the flow of control: conditional and unconditional branches and the associated test words.

17.1 Flags and Tests

TRUE and FALSE are built-in objects that are recognized by test words as flags for branching decisions. The following words create or combine flags:

```
AND ( flag1 flag2  --> flag )
  If flag1 and flag2 are both TRUE then TRUE else FALSE.

FALSE ( --> FALSE )
  Puts the FALSE flag on the stack.

FALSETRUE       ( --> FALSE TRUE )

FalseFalse      ( --> FALSE FALSE )

OR ( flag1 flag2  --> flag )
  If either flag1 or flag2 is TRUE then TRUE else FALSE.

ORNOT           ( flag1 flag2 --> flag3 )
  Logical OR followed by logical NOT.

NOT ( flag --> flag' )
  If flag is TRUE then FALSE else TRUE.

NOTAND          ( flag1 flag2 --> flag3 )
  Logical NOT, then logical AND.

ROTAND          ( flag1 ob flag2 --> ob flag3 )
  Does ROT, then logical AND.

TRUE            ( --> TRUE )
  Puts the TRUE flag on the stack.

TrueFalse       ( --> TRUE FALSE )

TrueTrue        ( --> TRUE TRUE )

XOR             ( flag1 flag2  --> flag )
  If both flag1 and flag2 are either TRUE or FALSE then FALSE, else TRUE.

COERCEFLAG      ( TRUE --> %1 )
                ( FALSE --> %0 )
  Converts a system flag to a real number flag.
```

17.1.1 General_Object_Tests

The following words test object type and equality:

```
EQ              ( ob1 ob2 --> flag )
  If objects ob1 and ob2 are the same object, i.e.  occupy the same
  physical space in memory, then TRUE else FALSE.

EQUAL           ( ob1 ob2 --> flag )
  where ob1 and ob2 are not primitive code objects. If objects ob1 and
  ob2 are the same then TRUE else FALSE (this word is the system
  equivalent of the user RPL command SAME)

2DUPEQ          ( ob1 ob2 --> ob1 ob2 flag )
  Returns TRUE if ob1 and ob2 have  the same physical address.

EQOR            ( flag1 ob1 ob2 --> flag2 )
  Does EQ, then logical OR.

EQUALOR         ( flag1 ob1 ob2 --> flag2 )
  Does EQUAL, the logical OR.

EQOVER          ( ob1 ob2 ob3 --> ob1 flag ob1 )
  Does EQ, then OVER.

EQUALNOT        ( ob1 ob2 --> flag )
  Returns FALSE if ob1 is equal to ob2.
```

The following words test an object's type. Words of the form TYPE...? have a stack diagram (ob --> flag); those of the form DTYPE...? or DUPTYPE...? duplicate the object first (ob --> ob flag).

Test Words	Object type
TYPEARRY? DTYPEARRY? DUPTYPEARRY?	array
TYPEBINT? DUPTYPEBINT?	binary integer
TYPECARRY?	complex array
TYPECHAR? DUPTYPECHAR?	character
TYPECMP? DUPTYPECMP?	complex number
TYPECOL? DTYPECOL? DUPTYPECOL?	program

TYPECSTR?   string
DTYPECSTR?
DUPTYPECSTR?

TYPEEXT?   unit
DUPTYPEEXT?

TYPEGROB?   graphics object
DUPTYPEGROB?

TYPEHSTR?   hex string
DUPTYPEHSTR?

TYPEIDNT?   identifier (global name)
DUPTYPEIDNT?

TYPELAM?   temporary identifier (local name)
DUPTYPELAM?

TYPELIST?   list
DTYPELIST?
DUPTYPELIST?

TYPERARRY?   real array

TYPEREAL?   real number
DTYPEREAL?
DUPTYPEREAL?

TYPEROMP?   ROM pointer (XLIB name)
DUPTYPEROMP?

TYPERRP?   Directory
DUPTYPERRP?

TYPESYMB?   Symbolic
DUPTYPESYMB?

TYPETAGGED?   Tagged
DUPTYPETAG?

17.1.2 Binary_Integer_Comparisons

The following words compare binary integers, returning TRUE or FALSE. Equality is tested in the sense of EQUAL (not EQ). Ordering treats all binary integers as unsigned. Some of these words are also available in combination with case words (see below).

#=   (# #' --> flag)   TRUE if # = #'.

#<>   (# #' --> flag)   TRUE if # <> #' (not equal).

#0=   (# --> flag)   TRUE if # = 0

```
#0<>     ( # --> flag )          TRUE if # <> 0

#<       ( # #' --> flag )       TRUE if # < #'

#>       ( # #' --> flag )       TRUE if # > #'

2DUP#<   ( # #' --> # #' flag ) TRUE if # < #'

2DUP#=   ( # #' --> # #' flag ) TRUE if # = #'

DUP#0=   ( # --> # flag )        TRUE if # = #0

DUP#1=   ( # --> # flag )        TRUE if # = #1

DUP#0<>  ( # --> # flag )        TRUE if # <> #0

DUP#1=   ( # --> # flag )        TRUE if # = #1

DUP#<7   ( # --> # flag )        TRUE if # < #7

DUP%0=   ( % --> % flag )        TRUE if % = %0

ONE#>    ( # --> flag )          TRUE if # > #1

ONE_EQ   ( # --> flag )          TRUE if # is ONE

OVER#>   ( # #' --> # flag )     TRUE if # > #'

OVER#0=  ( # ob --> # ob flag )  TRUE if # is #0

OVER#<   ( # #' --> # flag )     TRUE if # > #'

OVER#=   ( # #' --> # flag )     TRUE if # = #'

OVER#>   ( # #' --> # flag )     TRUE if # < #'
```

17.1.3 Decimal_Number_Tests

The following words compare real, extended real, and complex numbers, returning TRUE or FALSE.

```
%<       ( % %' --> flag )       TRUE if % < %'

%<=      ( % %' --> flag )       TRUE if % <= %'

%<>      ( % %' --> flag )       TRUE if % <> %'

%=       ( % %' --> flag )       TRUE if % = %'

%>       ( % %' --> flag )       TRUE if % > %'

%>=      ( % %' --> flag )       TRUE if % >= %'

%0<      ( % --> flag )          TRUE if % < 0
```

```
%0<>    ( % --> flag )          TRUE if % <> 0

%0=     ( % --> flag )          TRUE if % = 0

%0>     ( % --> flag )          TRUE if % > 0

%0>=    ( % --> flag )          TRUE if % >= 0


%%0<=   ( %% %%' --> flag )     TRUE if %% <= %%'

%%0<>   ( %% --> flag )         TRUE if %% <> 0

%%0=    ( %% --> flag )         TRUE if %% = 0

%%0>    ( %% --> flag )         TRUE if %% > 0

%%0>=   ( %% --> flag )         TRUE if %% >= 0

%%>     ( %% %%' --> flag )     TRUE if %% > %%'

%%>=    ( %% %%' --> flag )     TRUE if %% >= %%'

%%<=    ( %% %%' --> flag )     TRUE if %% <= %%'


C%%0=   ( C%% --> flag )        TRUE if C%% = (%%0,%%0)

C%0=    ( C% --> flag )         TRUE if C% = (0,0)
```

17.2 Words that Operate on the Runstream

In many cases, it is desirable to interrupt the normal threaded order of execution, and insert additional objects or skip others in the runstream. The following words are provided for these purposes.

' (--> ob)

This is the RPL analogue of the Lisp QUOTE and is one of the most fundamental control objects, allowing the evaluation of an object to be postponed. More precisely, assumes that the topmost body in the RUNSTREAM is non-empty, i.e. the interpreter pointer does not point at a SEMI; and (1) If the next object in the runstream is an object, then pushes this object onto the data stack and moves the interpreter pointer to the next object; (2) If the next object is an object pointer, then pushes the pointee on the data stack and similarly skips to the next object. As an example, evaluation of the secondaries

```
:: # 3 # 4 SWAP ;      and      :: # 3 # 4 ' SWAP EVAL ;
```

both produce the same result.

'R (--> ob)

If the object pointed to by the top pointer on the return stack (i.e. the first element in the second body in the runstream) is an object, then 'R pushes this object onto the data stack, and advances the pointer to the next object in the same composite. If the pointer points to an object pointer whose pointee is not SEMI, then pushes the pointee onto the data stack, and similarly advances the return stack pointer. If the pointee is SEMI, then If the first element in the second body in the runstream is an object pointer to SEMI, then pushes a null secondary onto the data stack and does not advance the return stack pointer. 'R is useful in defining prefix operators. For example, assume that PREFIXSTO is defined as :: 'R STO ; Then the sequence PREFIXSTO FRED ANOTHEROBJECT would first push FRED onto the data stack and then execute STO, after which execution resumes at ANOTHEROBJECT.

ticR (--> ob TRUE | FALSE)

This word works similarly to 'R, except that it returns a flag to indicate whether the end of the top return stack composite has been reached. That is, if the top return stack pointer points to an object pointer to SEMI, then ticR pops the return stack and returns only FALSE. Otherwise return the next object from the composite and TRUE, while advancing the return stack pointer to the next object.

>R (:: -->)

Inserts the body of :: into the runstream, just below the top one. (That is, pushes a pointer to the body of :: onto the return stack). An example of its use is

```
:: ' :: <foo> ; >R <bar> ;
```

which will, when executed, cause <bar> to be executed before <foo>.

R> (--> ::)

Creates a program object from the composite body pointed to by the top return stack pointer, and pushes the program on the data stack and pops the return stack. Example:

```
:: :: R> EVAL <foo> ; <bar> ;
```

which, when executed, will cause <bar> to be executed before <foo>.

R@ (--> ::)

Same as R> except that the return stack is not popped.

RDROP (-->)

Pops the return stack.

IDUP (-->)

Duplicates the top body in the runstream. (That is, pushes the RPL variable I onto the return stack).

COLA (-->)

Assuming that the interpreter pointer is pointing at an object other than SEMI, COLA drops the remainder of the program body past the object and executes the object. This provides for efficient tail recursion; the efficiency is gained in that COLA can be used to avoid excessive buildup of pending returns. An example of its use is in a definition of factorial:

```
        fact:         :: { LAM x } BIND # 1 factpair ABND
                      ;

        factpair:     :: LAM x #0= ?SEMI
                         LAM x #* LAM x #1- ' LAM x
                         STO COLA factpair
                      ;
```

In this example, the importance of COLA is in its occurrence before factpair in the definition of factpair. Without this use, computing n! would require n return stack levels, which, when the computation was completed, would merely be popped off (since their bodies would be empty). With the inclusion of COLA, the definition uses a fixed maximum number of levels, independent of the argument to the function.

?SEMI   (flag -->)

Exits the current program if flag is TRUE.

?SEMIDROP (ob TRUE -->) or (FALSE -->)

Drops ob if flag is TRUE; exits the current program if flag is FALSE.

?SKIP (flag -->)

If flag is TRUE, skips the next object following ?SKIP.

NOT?SEMI (flag -->)

Exits the current program if flag is FALSE.

17.3 If/Then/Else

The fundamental RPL if/then/else capability is provided by means of the words RPIT and RPITE:

RPITE (flag ob1 ob2 --> ?)

If flag is TRUE then drop flag and ob2 and EVALuate ob1, else drop flag and ob1 and EVALuate ob2. The RPL expression

```
' <foo> ' <bar> RPITE
```

is equivalent to the FORTH expression

```
IF <foo> ELSE <bar> THEN
```

RPIT (flag ob --> ?)

If flag is TRUE then drop flag and EVALuate ob, else just drop flag and ob. The RPL expression

```
' <foo> RPIT
```

is equivalent to the FORTH expression

```
IF <foo> THEN
```

However, prefix versions of these words are also available, and are more commonly used than the postfix forms:

IT (flag -->)

If flag is TRUE then execute the next object in the runstream; otherwise skip that object. For example,

```
DUPTYPEREAL? IT :: %0 %>C% ;
```

converts a real number to a complex number; does nothing if the argument is not a real number.

ITE (flag -->)

If flag is TRUE the execute the next object in the runstream, and skip the second object; otherwise skip the next object and execute the second. For example,

```
DUPTYPELIST? ITE INNERCOMP ONE
```

takes a list apart, leaving the count on the stack; for any other type of argument, push a binary integer #1 on the stack.

The converse of IT is

?SKIP (flag -->)

If flag is TRUE, skip the next object in the runstream; otherwise, execute it.

There is also an unconditional skip:

SKIP (-->)

Skips over the next object in the runstream and continues execution beyond it. The sequence SKIP ; is a NOP.

Combination Words:

Word	Stack	Equivalent
#0=ITE	(# -->)	#0= ITE
#<ITE	(# -->)	#0< ITE
#=ITE	(# -->)	#= ITE
#>ITE	(# -->)	#> ITE
ANDITE	(flag flag' -->)	AND ITE
DUP#0=ITE	(# --> #)	DUP #0= ITE
EQIT	(ob1 ob2 -->)	EQ IT
EQITE	(ob ob' -->)	EQ ITE
DUP#0=IT	(# --> #)	DUP #0= IT
SysITE		(# -->)
UserITE		(# -->)

17.4 CASE words

The word case is a combination of ITE, COLA and SKIP. That is, case takes a flag from the stack; if TRUE, case executes the object that follows it in the runstream while popping the return stack to the interpreter pointer, discarding the rest of the program that follows the object (like COLA). If FALSE, case skips the next object and continues with the program (like SKIP). For example, the following program executes different objects according to the value of a binary integer on the stack:

```
:: DUP #0= case ZEROCASE
   DUP ONE #= case ONECASE
   DUP TWO #= case TWOCASE
   ...
;
```

There are several words that contain case as part of their definitions. The above example can be written more compactly using OVER#=case:

```
:: ZERO OVER#=case ZEROCASE
   ONE OVER#=case ONECASE
   TWO OVER#=case TWOCASE
   ...
;
```

The actions of the words listed below are generally sufficiently clear from their names. The names have (up to) three parts: an initial part, then "case", then a final part. The initial part indicates what is done before the case action, i.e. "xxxcase..." is equivalent to "xxx case...". Words that have a final part after "case" are of two types. For one type, the final part indicates the conditionally executed object itself, i.e. "...caseyyy" is equivalent to "...case yyy." In the other type, the final part is a word or words that are incorporated into the following object. caseDROP and casedrop are of the first type and second type, respectively. caseDROP is equivalent to case DROP; casedrop is like case with a DROP incorporated into the next object. That is,

Words that COLA or SKIP the next object:

```
  #=casedrop    ( # # --> )
                ( # #' --> # )
        Should be named OVER#=casedrop.

  %1=case       ( % --> )

  %0=case       ( % --> flag )

  ANDNOTcase    ( flag1 flag2 --> )

  ANDcase       ( flag1 flag2 --> )

  case2drop     ( ob1 ob2 TRUE --> )
                ( FALSE --> )

  casedrop      ( ob TRUE --> )
                ( FALSE --> )

  DUP#0=case    ( # --> # )

  DUP#0=csedrp  ( # --> # ) # <> #0
                ( # -->   ) # = #0

  EQUALNOTcase  ( ob ob' --> )

  EQUALcase     ( ob ob' --> )
```

```
  EQUALcasedrp  ( ob ob' ob' --> )
                ( ob ob' ob'' --> ob )

  EQcase        ( ob1 ob2 --> )

  NOTcase       ( flag --> )

  NOTcasedrop   ( ob FALSE --> )
                ( TRUE --> )

  ORcase        ( flag1 flag2 --> )

  OVER#=case    ( # #' --> # )
```

Case words that either exit or continue with the next object:

```
  caseDoBadKey  ( flag --> ) Exit via DoBadKey

  caseDrpBadKey ( ob TRUE --> ) Exit via DoBadKey
                ( FALSE --> )

  case2DROP     ( ob1 ob2 TRUE --> )
                ( FALSE --> )

  caseDROP      ( ob TRUE --> )
                ( FALSE --> )

  caseFALSE     ( TRUE --> FALSE )
                ( FALSE --> )

  caseTRUE      ( TRUE --> TRUE )
                ( FALSE --> )

  casedrpfls    ( ob TRUE --> FALSE )
                ( FALSE --> )

  case2drpfls   ( ob1 ob2 TRUE --> FALSE )
                ( FALSE --> )

  casedrptru    ( ob TRUE --> TRUE )
                ( FALSE --> )

  DUP#0=csDROP  ( #0 -->  )
                ( #  --> # ) # <> 0.
  NOTcaseTRUE   ( FALSE --> TRUE )
                ( TRUE --> )
```

18. Stack Operations

The words listed in this chapter perform single or multiple stack operations.

```
2DROP           ( ob1 ob2 --> )
2DROP00         ( ob1 ob2 --> #0 #0 )
2DROPFALSE      ( ob1 ob2 --> FALSE )
2DUP            ( ob1 ob2 --> ob1 ob2 ob1 ob2 )
2DUP5ROLL       ( ob1 ob2 ob3 --> ob2 ob3 ob2 ob3 ob1 )
2DUPSWAP        ( ob1 ob2 --> ob1 ob2 ob2 ob1 )
2OVER           ( ob1 ob2 ob3 ob4 --> ob1 ob2 ob3 ob4 ob1 ob2 )
2SWAP           ( ob1 ob2 ob3 ob4 --> ob3 ob4 ob1 ob2 )
3DROP           ( ob1 ob2 ob3 --> )
3PICK           ( ob1 ob2 ob3 --> ob1 ob2 ob3 ob1 )
3PICK3PICK      ( ob1 ob2 ob3 --> ob1 ob2 ob3 ob1 ob2 )
3PICKOVER       ( ob1 ob2 ob3 --> ob1 ob2 ob3 ob1 ob3 )
3PICKSWAP       ( ob1 ob2 ob3 --> ob1 ob2 ob1 ob3 )
3UNROLL         ( ob1 ob2 ob3 --> ob3 ob1 ob2 )
4DROP           ( ob1 ob2 ob3 ob4 --> )
4PICK           ( ob1 ob2 ob3 ob4 --> ob1 ... ob4 ob1 )
4PICKOVER       ( ob1 ob2 ob3 ob4 --> ob1 ob2 ob3 ob4 ob1 ob4 )
4PICKSWAP       ( ob1 ob2 ob3 ob4 --> ob1 ob2 ob3 ob1 ob4 )
4ROLL           ( ob1 ob2 ob3 ob4 --> ob2 ob3 ob4 ob1 )
4UNROLL         ( ob1 ob2 ob3 ob4 --> ob4 ob1 ob2 ob3 )
4UNROLL3DROP    ( ob1 ob2 ob3 ob4 --> ob4 )
4UNROLLDUP      ( ob1 ob2 ob3 ob4 --> ob4 ob1 ob2 ob3 ob3 )
4UNROLLROT      ( ob1 ob2 ob3 ob4 --> ob4 ob3 ob2 ob1 )
5DROP           ( ob1 ... ob5 --> )
5PICK           ( ob1 ... ob5 --> ob1 ... ob5 ob1 )
5ROLL           ( ob1 ... ob5 --> ob2 ... ob5 ob1 )
5ROLLDROP       ( ob1 ... ob5 --> ob2 ... ob5 )
5UNROLL         ( ob1 ... ob5 --> ob5 ob1 ... ob4 )
6DROP           ( ob1 ... ob6 --> )
6PICK           ( ob1 ... ob6 --> ob1 ... ob6 ob1 )
6ROLL           ( ob1 ... ob6 --> ob2 ... ob6 ob1 )
7DROP           ( ob1 ... ob7 --> )
7PICK           ( ob1 ... ob7 --> ob1 ... ob7 ob1 )
7ROLL           ( ob1 ... ob7 --> ob2 ... ob7 ob1 )
8PICK           ( ob1 ... ob8 --> ob1 ... ob8 ob1 )
8ROLL           ( ob1 ... ob8 --> ob2 ... ob8 ob1 )
8UNROLL         ( ob1 ... ob8 --> ob8 ob1 ... ob7 )
DEPTH           ( ob1 ... obn ... --> #n )
DROP            ( ob --> )
DROPDUP         ( ob1 ob2 --> ob1 ob1 )
DROPFALSE       ( ob --> FALSE )
DROPNDROP       ( ... # ob ) Drops ob, then # objects
DROPONE         ( ob --> #1 )
DROPOVER        ( ob1 ob2 ob3 --> ob1 ob2 ob1 )
DROPRDROP       ( ob --> ) Drops ob, and pops 1 return stk level
DROPROT         ( ob1 ob2 ob3 ob4 --> ob2 ob3 ob1 )
DROPSWAP        ( ob1 ob2 ob3 --> ob2 ob1 )
DROPSWAPDROP    ( ob1 ob2 ob3 --> ob2 )
DROPTRUE        ( ob --> TRUE )
DROPZERO        ( ob --> #0 )
DUP             ( ob --> ob ob )
DUP#1+PICK      ( ... #n --> ... #n obn )
DUP3PICK        ( ob1 ob2 --> ob1 ob2 ob2 ob1 )
```

```
DUP4UNROLL       ( ob1 ob2 ob3 --> ob3 ob1 ob2 ob3 )
DUPDUP           ( ob --> ob ob ob )
DUPONE           ( ob --> ob ob #1 )
DUPPICK          ( ... #n --> ... #n obn-1 )
DUPROLL          ( ... #n --> ... #n obn-1 )
DUPROT           ( ob1 ob2 --> ob2 ob2 ob1 )
DUPTWO           ( ob --> ob ob #2 )
DUPUNROT         ( ob1 ob2 --> ob2 ob1 ob2 )
DUPZERO          ( ob --> ob ob #2 )
N+1DROP          ( ob ob1 ... obn #n --> )
NDROP            ( ob1 ... obn #n --> )
NDUP             ( ob1 ... obn #n --> ob1 ... obn ob1 ... obn )
NDUPN            ( ob #n --> ob ... ob #n )
ONEFALSE         ( --> #1 FALSE )
ONESWAP          ( ob --> #1 ob )
OVER             ( ob1 ob2 --> ob1 ob2 ob1 )
OVER5PICK        ( v w x y z --> v w x y z y v )
OVERDUP          ( ob1 ob2 --> ob1 ob2 ob1 ob1 )
OVERSWAP         ( ob1 ob2 --> ob2 ob1 ob1 )
OVERUNROT        ( ob1 ob2 --> ob1 ob1 ob2 )
PICK             ( obn ... #n --> ... obn )
ROLL             ( obn ... #n --> ... obn )
ROLLDROP         ( obn ... #n --> ... )
ROLLSWAP         ( obn ... ob #n --> ... obn ob )
ROT              ( ob1 ob2 ob3 --> ob2 ob3 ob1 )
ROT2DROP         ( ob1 ob2 ob3 --> ob2 )
ROT2DUP          ( ob1 ob2 ob3 --> ob2 ob3 ob1 ob3 ob1 )
ROTDROP          ( ob1 ob2 ob3 --> ob2 ob3 )
ROTDROPSWAP      ( ob1 ob2 ob3 --> ob3 ob2 )
ROTDUP           ( ob1 ob2 ob3 --> ob2 ob3 ob1 ob1 )
ROTOVER          ( ob1 ob2 ob3 --> ob2 ob3 ob1 ob3 )
ROTROT2DROP      ( ob1 ob2 ob3 --> ob3 )
ROTSWAP          ( ob1 ob2 ob3 --> ob2 ob1 ob3 )
SWAP             ( ob1 ob2 --> ob2 ob1 )
SWAP2DUP         ( ob1 ob2 --> ob2 ob1 ob2 ob1 )
SWAP3PICK        ( ob1 ob2 ob3 --> ob1 ob3 ob2 ob1 )
SWAP4PICK        ( ob1 ob2 ob3 ob4 --> ob1 ob2 ob4 ob3 ob4 )
SWAPDROP         ( ob1 ob2 --> ob2 )
SWAPDROPDUP      ( ob1 ob2 --> ob2 ob2 )
SWAPDROPSWAP     ( ob1 ob2 ob3 --> ob3 ob1 )
SWAPDROPTRUE     ( ob1 ob2 --> ob2 TRUE )
SWAPDUP          ( ob1 ob2 --> ob2 ob1 ob1 )
SWAPONE          ( ob1 ob2 --> ob2 ob1 #1 )
SWAPOVER         ( ob1 ob2 --> ob2 ob1 ob2 )
SWAPROT          ( ob1 ob2 ob3 --> ob3 ob2 ob1 )
SWAPTRUE         ( ob1 ob2 --> ob2 ob1 TRUE )
UNROLL           ( ... ob #n --> ob ... )
UNROT            ( ob1 ob2 ob3 --> ob3 ob1 ob2 )
UNROT2DROP       ( ob1 ob2 ob3 --> ob3 )
UNROTDROP        ( ob1 ob2 ob3 --> ob3 ob1 )
UNROTDUP         ( ob1 ob2 ob3 --> ob3 ob1 ob2 ob2 )
UNROTOVER        ( ob1 ob2 ob3 --> ob3 ob1 ob2 ob1 )
UNROTSWAP        ( ob1 ob2 ob3 --> ob3 ob2 ob1 )
ZEROOVER         ( ob --> ob #0 ob )
reversym         ( ob1 ... obn #n --> obn ... ob1 #n )
```

19. Memory Operations

The words presented in this chapter manipulate directories, variables, and system ram.

19.1 Temporary Memory

The user word NEWOB creates a new copy of an object in temporary memory. There are a few internal variations on this theme:

```
CKREF          ( ob --> ob' )
                 If ob is in TEMPOB, is not embedded
                 in a composite object, and is not
                 referenced, then does nothing. Otherwise
                 copies ob to TEMPOB and returns the copy.

INTEMNOTREF?   ( ob --> ob flag )
                 If the input object is in TEMPOB area,
                 is not embedded in a composite object,
                 and is not referenced, returns ob and
                 TRUE, otherwise returns ob and FALSE.

TOTEMPOB       ( ob --> ob' )
                 Copies ob into TEMPOB and returns pointer
                 to the new ob.
```

19.2 Variables and Directories

The system RPL basis of user STO and RCL is the words STO, CREATE, and @:

```
CREATE ( ob id --> )
  Creates a RAM-WORD with ob as its object part and the NAME
  FORM from id as its name part, in the current directory.
  An error occurs if ob is or contains the current directory
  ("Directory Recursion"). Assumes that ob is not a
  primitive code object.

STO ( ob id --> )
    ( ob lam --> )
  In the lam case, the temporary identifier lam is re-bound
  to ob.  The binding is to the first such temporary
  identifier object matching lam in the Temporary
  Environment area (searching from the first temporary
  environment to the last). An error is returned if lam is
  unbound.  In the id case, STO attempts to match id to the
  name part of a global variable.  If resolution is
  unsuccessful, STO creates a variable with ob as its object
  part and the name form from id as its name part, in the
  current directory. If resolution is successful, then ob
  replaces the object part of the resolved variable. If any
  updatable system object pointers reference the object part
  of the resolved variable, then the object part is placed
  into the temporary object area prior to the replacement
  and all affected updatable system object pointers are
```

adjusted to reference the copy of the object part in the temporary object area. For the id case, STO assumes that ob is not a primitive code object.

```
@   ( id --> ob TRUE )
    ( id --> FALSE )
    ( lam --> ob TRUE )
    ( lam --> FALSE )
```

In the lam case, @ attempts to match lam to the temporary identifier object part of a binding in the Temporary Environment area (searching from the first temporary environment to the last). If successful, then the object bound to lam is returned along with a TRUE flag; else, a FALSE flag is returned. In the id case, @ attempts to match id to the name part of a global variable, starting in the current directory, and working up through parent directories if necessary. If the resolution is unsuccessful, then a FALSE flag is returned. Otherwise, the object part of the resolved variable is returned with a TRUE flag.

One difficulty in using STO and @ is that they make no distinctions for built-in commands; with SIN as its (object) argument, STO will blithely copy the entire body of SIN into a variable. @ then would recall that undecompilable program. For this reason, it is better to use SAFESTO and SAFE@, which work like STO and @ except that they automatically convert ROM bodies into XLIB names (SAFESTO) and back again (SAFE@).

Additional extensions are:

```
?STO_HERE ( ob id --> )
          ( ob lam --> )
```

This is the system version of user STO. It is the same as SAFESTO, except that for global variables, it a) stores only in the current directory; and b) will not overwrite a stored directory.

```
SAFE@_HERE ( id --> ob TRUE )
           ( id --> FALSE )
           ( lam --> ob TRUE )
           ( lam --> FALSE )
```

Same as SAFE@, except for global variables the search is restricted to the current directory.

Other related words:

PURGE (id -->) Purges variable specified by id; does no type check on stored object.

XEQRCL (id --> ob) Same as SAFE@ for global variables, but errors if variable is nonexistent

XEQSTOID (ob id -->) Alternate name for ?STO_HERE

19.2.1 Directories A directory (abbreviated "rrp" from its original name "ramrompair") is an object whose body contains a linked-list of global variables--named objects referenced by global names. The body also contains a library ID number that associates ("attaches") a library object with the directory so that the library's commands follow the directory's variables in the name compilation search order.

A directory may be "rooted", i.e. stored in a global variable (which may be within another directory body), in which case its variable's names are available for compilation. The particular directory in which a name resolution search begins is called the "current directory," or the "context directory;" this directory is specified by the contents of a system RAM location. An unrooted directory (in tempob or in a port, for example), should never be selected as the context directory. Nor can there be any references within a directory in tempob; a directory is not a composite object, so garbage collection cannot work properly if such references exist. For this reason, an internally referenced directory should not be removed by PURGE--use XEQPGDIR instead.

In addition to the context, another system RAM location identifies the "stopsign" directory, which is acts as the ending point for a name resolution search much as the context directory is the starting point. By using the stopsign, you can restrict name resolution searches to a specific range; however, you should use error traps to insure that the stopsign is reset to the home directory when an error occurs.

The home directory (aka "sysramrompair") is the default for both context and stopsign. This is not a normal directory, in that it is never unrooted, and contains additional structure that ordinary directories don't have (such as multiple library attachments and alternate message and command hash tables).

A directory is a data-class object so that execution of a directory merely returns it to the stack. However, global name execution has the property that executing the name of a rooted (stored) directory makes that directory the current directory rather than executing the directory itself.

The following words are available for directory manipulation:

CONTEXT! (rrp -->)
  Stores a pointer to a rooted directory as the current directory

CONTEXT@ (--> rrp)
  Recalls the current context directory.

CREATEDIR (id -->)
  Creates a directory object in the current directory.

DOVARS (--> { id1 id2 ... })
Returns list of variable names from the current directory.

HOMEDIR (-->)
Makes HOME the current directory.

PATHDIR (--> { HOME dir dir ... })
Returns the current path.

UPDIR (-->)
Switches context to the parent of the current directory.

XEQORDER ({ id1 id2 ... } -->)
ORDERs current directory.

XEQPGDIR (id -->)
Purges a directory while respecting reference/garbage collection conventions.

19.3 The Hidden Directory

There is a hidden, nullnamed directory at the beginning of the home directory, that contains the user key definitions and alarm information. Application programs may use this directory as well. However, remember that the user has no way to detect or remove variables from this directory, so an application should either remove such variables before finishing, or to provide a command that lets the user remove specific variables from the hidden directory.

These words provide store, recall and purge capabilities for the hidden directory:

PuHiddenVar (id -->) Purges the hidden variable named id.

```
RclHiddenVar ( id --> ob TRUE )
             ( id --> FALSE   )
```
Recalls (@) a hidden variable.

StoHiddenVar (ob id -->)
Stores ob in hidden variable

19.4 Additional Memory Utilities

GARBAGE (-->)
  Forces garbage collection.

MEM(--> #)
  Returns the amount of free memory (a garbage collection is not forced)

OCRC (ob --> #nibbles checksum(hxs) Returns size of object in nibbles and
  a hex string checksum

getnibs (hxsDATA hxsADDR --> hxsDATA') Internal RPL version of PEEK

putnibs (hxsDATA hxsADDR -->) Internal RPL version of POKE

20. Display Management & Graphics

Most user RPL graphics commands are directed to the graphics screen, which is the graphics object visible in the plot environment. However, the "text screen," the grob visible in the standard stack environment, has the same properties as the graph screen, and should be used by application programs for graphics displays whenever possible, to leave the graph screen as a user "owned" resource. The EquationWriter does this, for example, as does the HP82211A HP Solve Equation Library card.

20.1 Display Organization

HP 48 system RAM contains three dedicated graphics objects used for display purposes:

```
   Pointer           Grob         Location
  ---------    ----------------  ---------

               +---------------+
  HARDBUFF2 -> | Menu labels   |  (Low Mem)
               +---------------+
  ABUFF     -> | text  grob    |
               +---------------+
  GBUFF     -> | graph grob    |  (Hi  Mem)
               +---------------+
```

The text grob and graph grob may be enlarged, and may be scrolled.

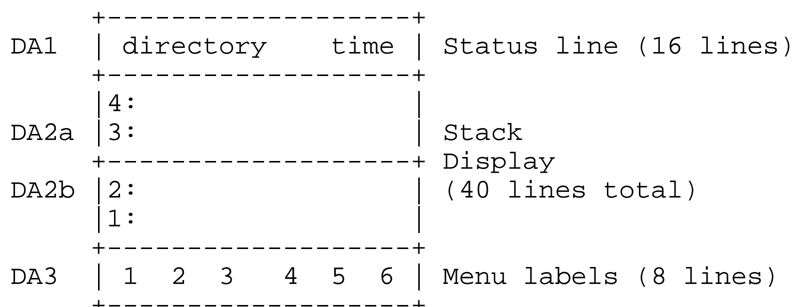
The word TOADISP switches makes the text grob visible; TOGDISP switches the LCD to the graph grob.

The following words are useful for returning display grobs to the stack:

```
ABUFF  ( --> textgrob )
GBUFF  ( --> graphgrob )
HARDBUFF  ( --> HBgrob )
  Returns whichever of the text or graph grob is currently displayed.
HARDBUFF2 ( --> menugrob )
HBUFF_X_Y( --> HBgrob #x1 #y1 )
```

A ram pointer named VDISP indicates which grob is currently shown in the display. VDISP may point to either the text grob or the graph grob. VDISP is not directly accessible - the word HARDBUFF returns the current display grob to the stack (see below). Remember that ABUFF and GBUFF just return pointers, so if the grob is being recalled for modification and later return to the user, TOTEMPOB should be used to create a unique copy in temporary memory.

From a user's point of view, the text display is organized into three regions, and the internal numbering convention for these areas is reflected in many of the display control words (see "Display Area Control" below). The display areas are numbered 1, 2, and 3. The letters "DA", for "Display Area", are found in the names of some display control words.



Display area 2 is actually divided into areas 2a and 2b, a distinction most often used by the command line line. The boundary between 2a and 2b can move, but the overall sizes of areas 1, 2, and 3 are fixed.

20.2 Preparing the Display

Two words establish control over the text display. These words are RECLAIMDISP and ClrDA1IsStat.

The word RECLAIMDISP performs the following actions:

+ Makes sure the text grob is the current display.

+ Clears the text display.

+ Resizes the text grob to the standard size (131 wide by 56 high) if necessary.

RECLAIMDISP is very much like the user word CLLCD, except that CLLCD does not resize the text grob.

The word ClrDA1IsStat suspends the ticking clock display, and is optional. If user input will be requested using words like WaitForKey or a parameterized outer loop (see "Keyboard Input"), then the clock updates will continue, and may botch the display.

An example usage of ClrDA1IsStat can be found in the Periodic Table application, where a user can enter a molecular formula. The word WaitForKey is used to get keystrokes, and ClrDA1IsStat prevents the clock from overwriting the Periodic Table grid display.

If the menu display is not needed, the word TURNMENUOFF will remove DA3 from the display and enlarge the text grob to be

131x64. The corresponding word TURNMENUON restores the menu display.

A simplified framework for an application secondary which can be invoked by an end user and uses the text display looks like this:

```
::
  ClrDA1IsStat          ( *Suspend clock display updates* )
  RECLAIMDISP           ( *Assert & clear alpha display* )
  TURNMENUOFF           ( *Remove menu keys* )

  < application >

  ClrDAsOK   -\         ( *Tell the 48 to redraw the lcd* )
    -or-      > Choose one
  SetDAsTemp -/         ( *Freeze all display areas* )
;
```

20.3 Controlling Display Refresh

When an application terminates or returns to the system outer loop for keyboard input, several internal versions of the user word FREEZE are available to control the display, and there is a word that ensures that certain display or all display areas will be redrawn:

```
SetDA1Temp      Freeze display area 1
SetDA2aTemp     Freeze display area 2a
SetDA2bTemp     Freeze display area 2b
SetDA3Temp      Freeze display area 3
SetDAsTemp      Freeze all display areas
ClrDAsOK        Redraw the entire lcd when program ends
```

There are still more variations on this theme - see the chapter "Keyboard Input" for more.

20.4 Clearing the Display

The following words may be used to clear either the whole display or a portion of HARDBUFF. Remember that HARDBUFF refers to the currently displayed grob, which is either the text grob or the graph grob.

```
BLANKIT         ( #startrow #rows --> )
                  Clears #rows starting at #startrow
BlankDA12       ( --> )
                  Clears rows 0 through 56
BlankDA2        ( --> )
                  Clears rows 16 through 56
CLEARVDISP      ( --> )
                  Zeros out all of HARDBUFF
Clr16           ( --> )
                  Clears top 16 pixel rows
Clr8            ( --> )
                  Clears top 8 pixel rows
Clr8-15         ( --> )
                  Clears pixel rows 8-15
```

20.5 Annunciator Control

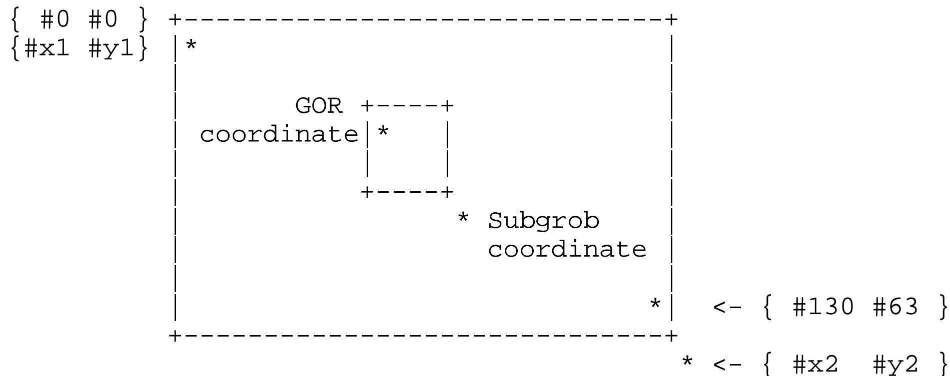
The following words control the left-shift, right-shift, and alpha annunciators. It is unlikely that an application should have to control these directly, and misuse of these words can lead to misleading displays after an application terminates.

```
ClrAlphaAnn     Clears the alpha annunciator
ClrLeftAnn      Clears the left-shift annunciator
ClrRightAnn     Clears the right-shift annunciator
SetAlphaAnn     Sets the alpha annunciator
SetLeftAnn      Sets the left-shift annunciator
SetRightAnn     Sets the right-shift annunciator
```

20.6 Display Coordinates

The upper-left pixel of the display has the coordinates x=0 y=0, which are the same as user pixel coordinates { #0 #0 }. The lower-right pixel coordinate is x=130 y=63.

NOTE: subgrobs are taken from the upper-left coordinate to the pixel below and to the right of the lower right corner. The terms #x1 and #y1 refer to the upperleft pixel of a sub area, while #x2 and #y2 refer to the pixel below and the right of the lower right corner.



20.6.1 Window_Coordinates

The following routines return HARDBUFF and coordinates for portions of the display in a form suitable for a subsequent call to SUBGROB. The terms #x1 and #y1 refer to the upper left corner of the window on the currently displayed grob. If the grob has been scrolled, these will NOT be #0 #0!

If HARDBUFF has been scrolled, some display words may not be appropriate to use since they depend on the upper left corner of the display being #0 #0. The LCD is then called the "window", and the terms #x1 and #y1 will refer to the pixel coordinates of the upper left corner of the window. The word HBUFF_X_Y returns HARDBUFF and these window coordinates. The word WINDOWCORNER returns just the window coordinates. The words DISPROW1* and DISPROW2* mentioned below work relative to the window corner.

```
Rows8-15        ( --> HBgrob #x1 #y1+8 #x1+131 #y1+16 )
TOP16           ( --> HBgrob #x1 #y1 #x1+131 #y1+16 )
TOP8            ( --> HBgrob #x1 #y1 #x1+131 #y1+8 )
WINDOWCORNER    ( --> #x #y )
                  Returns pixel numbers of upperleft corner
                  of the window
```

The word Save16 calls TOP16 and SUBGROB to produce a grob consisting of the top 16 rows of the current display:

```
Save16          ( --> grob )
```

Equivalent words that save the top eight rows or rows 8-15 are not in the HP 48, but can be written as follows:

```
:: TOP8 SUBGROB ; ( --> grob ) ( *Saves the top 8 rows* )
:: TOP8-15 SUBGROB ; ( --> grob ) ( *Saves the top 8-15 rows* )
```

20.7 Displaying Text

There are three fonts available in the HP 48, distinguished by size. The smallest font is variable width; the medium and large fonts are fixed width.

The words described below display text using the medium and large fonts in specific areas. Use of the small fonts, and other placement options for the medium and large fonts must be done in graphics, which is described later.

20.7.1 Standard_Text_Display_Areas

When the text grob is the current display AND has not been scrolled, the following words may be used to display text in the medium (5x7) font. Long strings are truncated to 22 characters with a trailing ellipsis (...), and strings shorter than 22 characters are blank filled.

```
DISPROW1        ( $ --> )
DISPROW2        ( $ --> )
DISPROW3        ( $ --> )
DISPROW4        ( $ --> )
DISPROW5        ( $ --> )
DISPROW6        ( $ --> )
DISPROW7        ( $ --> )
DISPN           ( $ #row --> )
DISP5x7         ( $ #start #max )
```

```
                        (0,0)                (130,0)
   DISPROW1 writes into   +--------------------+
                          |                    |
                          +--------------------+
                        (0,7)                 (130,7)

                        (0,8)                 (130,8)
   DISPROW2 writes into   +--------------------+
                          |                    |
                          +--------------------+
                        (0,15)                (130,15)
         (etc.)
```

The word DISP5x7 may be used to display a string that spans more than one line of the display. The string must have embedded carriage returns to show where to break to the next display line. If a line segment is greater than 22 characters, it will be truncated and displayed with a trailing ellipsis (...). The string is displayed starting at row #start for #max rows.

The following words may be used to display text in the large (5x9) font. Long strings are truncated to 22 characters with a trailing ellipsis (...), and strings shorter than 22 characters are blank filled.

```
BIGDISPROW1     ( $ --> )
BIGDISPROW2     ( $ --> )
BIGDISPROW3     ( $ --> )
BIGDISPROW4     ( $ --> )
BIGDISPN        ( $ #row --> )


                        (0,17)             (130,0)
BIGDISPROW1 writes into   +---------------------+
                          |                     |
                          +---------------------+
                        (0,26)             (130,26)

                        (0,27)             (130,27)
BIGDISPROW2 writes into   +---------------------+
                          |                     |
                          +---------------------+
                        (0,36)             (130,36)
         (etc.)
```

20.7.2 Temporary_Messages

Sometimes it is convenient to display a warning, then return the display to its previous state. There are several techniques and tools available for this. The easiest way to do this is with the word FlashWarning. The code for FlashWarning looks like this:

```
FlashWarning            ( $ --> )
::
  ERRBEEP               ( *Generate an error beep* )
  Save16                ( *Save the top 16 pixel rows* )
  SWAP DISPSTATUS2      ( *Display the warning* )
  VERYVERYSLOW          ( *Wait about 3 seconds* )
  Restore16             ( *Restore the top 16 rows* )
;
```

Variations on FlashWarning can be constructed using words like TOP16 or a version suggested above that saves fewer rows. The example below saves the top 8 rows and displays a one line message for about .6 second:

```
::
  TOP8 SUBGROB          ( *Save the top 8 rows* )
  SWAP DISPROW1*        ( *Display the message* )
  VERYSLOW VERYSLOW     ( *Short delay* )
  Restore8              ( *Restore the top 8 rows* )
;
```

NOTE: It is important to use DISPROW1* and DISPROW2* instead of DISPROW1 and DISPROW2 if there is any chance that HARDBUFF has been scrolled. There are no corresponding words for other display lines.

20.8 Graphics Objects

The following section presents tools for creating, manipulating, and displaying graphics objects.

20.8.1 Warnings

Here are two warnings:

1. The term "bang-type operation" refers to an operation performed directly upon an object without making a copy. The naming convention for words which perform this kind of operation often have an exclamation point to denote a "bang" operation, such as GROB! or GROB!ZERO.

   You must remember two things when using "bang" operations:

   + Since the object itself is modified, any pointers on the stack that refer to that object will now point to a changed object. The word CKREF may be used to ensure that an object is unique.

   + These operations have no error checking, so improper or out-of-range parameters may corrupt memory beyond recovery.

2. In practice, it is best to use the word XYGROBDISP to place a grob into the display grob. The word XYGROBDISP is conservative in nature - if the graphic to be placed in HARDBUFF would exceed the boundaries of HARDBUFF, the HARDBUFF grob is enlarged to accomodate the new grob.

20.8.2 Graphics_Tools

The following words create or modify graphics objects:

```
$>BIGGROB       ( $ --> grob ) ( 5x9 font )
$>GROB          ( $ --> grob ) ( 5x7 font )
$>grob          ( $ --> grob ) ( 3x7 font )
DOLCD>          ( --> 64x131grob )
GROB!           ( grob1 grob2 #col #row --> )
                  Stores grob1 into grob2. This is a
                  bang-type word with no error checks!
GROB!ZERO       ( grob #x1 #y1 #x2 #y2 --> grob' )
                  Zeros out a rectangular section of a
                  grob.  NOTE: Bang-type operation.
GROB!ZERODRP    ( grob #x1 #y1 #x2 #y2 --> )
                  Zeros out a rectangular section of a
                  grob.  NOTE: Bang-type operation!
GROB>GDISP      ( grob --> )
                  Stores graph grob with new grob.
HARDBUFF        ( --> HBgrob (the current display grob) )
HEIGHTENGROB    ( grob #rows --> )
                  Adds #rows to grob, unless grob is null.
                  NOTE: Assumes text grob or graph grob!
INVGROB         ( grob --> grob' )
                  Invert grob data bits - bang-type.
LINEOFF         ( #x1 #y1 #x2 #y2 --> )
                  Clears pixels in a line in text grob
                  Note: #x2 must be > #x1 (use ORDERXY#)
LINEOFF3        ( #x1 #y1 #x2 #y2 --> )
                  Clears pixels in a line in graph grob
                  Note: #x2 must be > #x1 (use ORDERXY#)
LINEON          ( #x1 #y1 #x2 #y2 --> )
                  Sets pixels in a line in text grob
                  Note: #x2 must be > #x1 (use ORDERXY#)
LINEON3         ( #x1 #y1 #x2 #y2 --> )
                  Sets pixels in a line in graph grob
                  Note: #x2 must be > #x1 (use ORDERXY#)
MAKEGROB        ( #height #width --> grob )
ORDERXY#        ( #x1 #y1 #x2 #y2 --> #x1 #y1 #x2 #y2 )
                  Orders two points for line drawing
PIXOFF          ( #x #y --> )
                  Clears a pixel in the text grob
PIXOFF3         ( #x #y --> )
                  Clears a pixel in the graph grob
PIXON           ( #x #y --> )
                  Sets a pixel in the text grob
PIXON?          ( #x #y --> flag )
                  Returns TRUE if text grob pixel is set
PIXON?3         ( #x #y --> flag )
                  Returns TRUE if graph grob pixel is set
PIXON3          ( #x #y --> )
                  Sets a pixel in the graph grob
SUBGROB         ( grob #x1 #y1 #x2 #y2 --> subgrob )
Symb>HBuff      ( symb --> )
                  Displays symb in HARDBUFF in Equation-
                  Writer form.  May enlarge HARDBUFF, so
                  do RECLAIMDISP afterwards.
```

```
TOGLINE          ( #x1 #y1 #x2 #y2 --> )
                   Toggles pixels in a line in text grob
TOGLINE3         ( #x1 #y1 #x2 #y2 --> )
                   Toggles pixels in a line in graph grob
```

NOTE: #x2 must be greater than #x1 for line drawing!

20.8.3 Grob_Dimensions

The following words return or verify size information:

```
CKGROBFITS       ( grob1 grob2 #n #m --> grob1 grob2' #n #m )
                   Truncates grob2 if it doesn't fit in grob1
DUPGROBDIM       ( grob --> grob #height #width )
GBUFFGROBDIM     ( --> #height #width )
                   Returns dimensions of graph grob
GROBDIM          ( grob --> #height #width )
GROBDIMw         ( grob --> #width )
```

20.8.4 Built-in_Grobs

The following words refer to built-in grobs:

```
BigCursor        5x9 Cursor (outline box)
CROSSGROB        5x5 "+" symbol
CURSOR1          5x9 Insert Cursor (arrow)
CURSOR2          5x9 Replace Cursor (solid box)
MARKGROB         5x5 "X" symbol
MediumCursor     5x7 Cursor (outline box)
SmallCursor      3x5 Cursor (outline box)
```

20.8.5 Menu_Display_Utilities

Menu labels are grobs which are 8 rows high and 21 pixels wide. The columns for menu key labels in HARDBUFF2 are:

```
        ZERO            Softkey 1
        TWENTYTWO       Softkey 2
        # 0002C         Softkey 3
        # 00042         Softkey 4
        # 00058         Softkey 5
        # 0006E         Softkey 6
```

The routine DispMenu.1 redisplays the current menu; the routine DispMenu redisplays the current menu and also calls SetDA3Valid to "freeze" the menu display line.

The following words convert objects to menu labels and display the labels at the given column number:

```
Grob>Menu       ( #col 8x21grob --> )
                  Displays an 8x21 (only!) grob
Id>Menu         ( #col Id --> )
                  Recalls Id and displays standard label
                  or directory label, depending on the
                  contents of Id.
Seco>Menu       ( #col seco --> )
                  Evaluates secondary and uses result to
                  produce and display appropriate menu label
Str>Menu        ( #col $ --> )
                  Makes and displays standard menu label
```

The following words convert strings to the different kinds of available menu key grobs:

```
MakeBoxLabel    ( $ --> grob ) Box with bullet in it
MakeDirLabel    ( $ --> grob ) Box with directory bar
MakeInvLabel    ( $ --> grob ) White label (solver)
MakeStdLabel    ( $ --> grob ) Black label (standard)
```

20.9 Scrolling the Display

The following words are available for scrolling the display:

```
SCROLLDOWN      ( *Scroll down one pixel with repeat* )
SCROLLLEFT      ( *Scroll left one pixel with repeat* )
SCROLLRIGHT     ( *Scroll right one pixel with repeat* )
SCROLLUP        ( *Scroll up one pixel with repeat* )

JUMPBOT         ( *Move window to bottom edge of grob* )
JUMPLEFT        ( *Move window to left edge of grob* )
JUMPRIGHT       ( *Move window to right edge of grob* )
JUMPTOP         ( *Move window to bottom edge of grob* )
```

The SCROLL* words test to see if their corresponding arrow key is being held down, and repeat their action until the

edge of the grob is reached or the key released.

The example below illustrates a series of graphics operations and the use of a Parameterized Outer Loop which provides scrolling for the user.

```
*--------------------------------------------------------

*
* Include the header file KEYDEFS.H, which defines words
* like kcUpArrow at physical key numbers.
*
INCLUDE KEYDEFS.H
*
* Include the eight characters needed for binary download
*
ASSEMBLE
        NIBASC  /HPHP48-D/
RPL
*
* Begin the secondary
*
::
  RECLAIMDISP            ( *Claim the alpha display* )
  ClrDA1IsStat           ( *Temporarily disable clock* )
*                        ( *Try removing ClrDA1IsStat* )
  ZEROZERO               ( #0 #0 )
  150 150 MAKEGROB       ( #0 #0 150x150grob )
  XYGROBDISP             (  )
  TURNMENUOFF            ( *Turn off menu line* )
*
* Draw diagonal lines.  Remember that LINEON requires
* requires #x2>#x1!
*
  ZEROZERO               ( #x1 #y1 )
  149 149                ( #x1 #y1 #x2 #y2 )
  LINEON                 ( *Draw line* )
  ZERO 149               ( #x1 #y1 )
  149 ZERO               ( #x1 #y1 #x2 #y2 )
  LINEON                 ( *Draw line* )
*
* Place text
*
  HARDBUFF
  75 50 "SCROLLING"      ( HBgrob 75 150 "SCROLLING" )
  150 CENTER$3x5         ( HBgrob )
  75 100 "EXAMPLE"       ( HBgrob 75 100 "EXAMPLE" )
  150 CENTER$3x5         ( HBgrob )
  DROPFALSE              ( FALSE )
  { LAM Exit } BIND      ( *Bind POL exit flag* )
  ' NOP                  ( *No display action*  )
  ' ::                   ( *Hard key handler*   )
    kpNoShift #=casedrop
      ::
        kcUpArrow    ?CaseKeyDef
                         :: TakeOver SCROLLUP ;
```

```
          kcLeftArrow  ?CaseKeyDef
                           :: TakeOver SCROLLLEFT ;
          kcDownArrow  ?CaseKeyDef
                           :: TakeOver SCROLLDOWN ;
          kcRightArrow ?CaseKeyDef
                           :: TakeOver SCROLLRIGHT ;
          kcOn         ?CaseKeyDef
                           :: TakeOver
                              TRUE ' LAM Exit STO ;
          kcRightShift   #=casedrpfls
          DROP 'DoBadKeyT
       ;
     kpRightShift #=casedrop
       ::
          kcUpArrow    ?CaseKeyDef
                           :: TakeOver JUMPTOP ;
          kcLeftArrow  ?CaseKeyDef
                           :: TakeOver JUMPLEFT ;
          kcDownArrow  ?CaseKeyDef
                           :: TakeOver JUMPBOT ;
          kcRightArrow ?CaseKeyDef
                           :: TakeOver JUMPRIGHT ;
          kcRightShift #=casedrpfls
          DROP 'DoBadKeyT
       ;
     2DROP 'DoBadKeyT
   ;
   TrueTrue              ( *Key control flags* )
   NULL{}                ( *No softkeys here*  )
   ONEFALSE              ( *1st row, no suspend* )
   ' LAM Exit            ( *App exit condition* )
   ' ERRJMP              ( *Error handler* )
   ParOuterLoop          ( *Run the ParOuterLoop* )
   TURNMENUON            ( *Restore menu row* )
   RECLAIMDISP           ( *Resize and clear display* )
;
```

The above code, if stored in a file SCROLL.S, can be compiled as follows:

```
RPLCOMPILE SCROLL.S
SASM SCROLL.A
SLOAD -H SCROLL.M
```

This example also assumes that the file KEYDEFS.H is either in the same directory or the source file has been modified to reflect the location of KEYDEFS.H. The loader control file SCROLL.M looks like this:

```
OU SCROLL
LL SCROLL.LR
SU XR
SE ENTRIES.O
RE SCROLL.O
```

The final file, SCROLL, may be binary downloaded to the HP 48 for a test.

When SCROLL is running, the arrow keys scroll the display, and the right-shifted arrow keys move the window to the corresponding boundary. The [ATTN] key terminates the program.

For more details on the ParOuterLoop, see the chapter "Keyboard Control".

21. Keyboard Control

A program that requires keyboard input from the user may choose from three basic techniques available with internal RPL, listed in order of increasing complexity:

1. Wait for an individual keystroke, then decide what to do with it.
2. Call the internal form of INPUT.
3. Set up a Parameterized Outer Loop to control an entire application environment.

The following sections discuss the internal numbering scheme for keys and each of the above three key processing strategies.

21.1 Key Locations

The user word WAIT returns a real number which is encoded in the form rc.p, where:

r = The row of the key
c = The column of the key
p = The shift plane.

p	Primary Planes	p	Alpha Planes
0 or 1	Unshifted	4	Alpha
2	Left-shifted	5	Alpha left-shifted
3	Right-shifted	6	Alpha right-shifted

Internally, key locations are represented with two binary integers: #KeyCode, which defines a physical key, and #Plane, which defines the shift plane.

The file KEYDEFS.H, supplied with the RPL compiler, defines the following terms for key planes:

```
DEFINE  kpNoShift        ONE
DEFINE  kpLeftShift      TWO
DEFINE  kpRightShift     THREE
DEFINE  kpANoShift       FOUR
DEFINE  kpALeftShift     FIVE
DEFINE  kpARightShft     SIX
```

Keys are numbered internally from 1 to 49, starting at the upper left corner of the keyboard. Primary key definitions are also provided in KEYDEFS.H. Here are a few of them:

```
DEFINE  kcMenuKey1       ONE
DEFINE  kcMenuKey2       TWO
DEFINE  kcMenuKey3       THREE
DEFINE  kcMenuKey4       FOUR
DEFINE  kcMenuKey5       FIVE
DEFINE  kcMenuKey6       SIX
DEFINE  kcMathMenu       SEVEN
DEFINE  kcPrgmMenu       EIGHT
DEFINE  kcCustomMenu     NINE
     ...
DEFINE  KcPlus           FORTYNINE
```

The use of these definitions in source code is encouraged for legibility.

The translation between internal key numbering and rc.p numbering may be carried out with two words:

```
Ck&DecKeyLoc    ( %rc.p --> #KeyCode #Plane )
CodePl>%rc.p    ( #KeyCode #Plane --> %rc.p )
```

21.2 Waiting for a Key

If an application needs to wait for a single key, such as a yes-no-attn type decision, it is best to use the word WaitForKey, which returns a fully formed keystroke. WaitForKey also keeps the HP 48 in a low-power state until a key is pressed and handles the alpha and shift annunciators and alarm processing.

The following words are available:

```
CHECKKEY        ( --> #KeyCode TRUE )
                ( --> FALSE )
                  Returns, but does not pop, the next
                  key in the buffer.
FLUSHKEYS       ( --> )
                  Flush the key buffer.
GETTOUCH        ( --> #KeyCode TRUE )
                ( --> FALSE )
                  Pops next key from buffer.
KEYINBUFFER?    ( --> FLAG )
                  Returns TRUE if a key is in the buffer,
                  otherwise returns FALSE.
ATTN?           ( --> flag )
                  Returns TRUE if [ATTN] has been pressed
ATTNFLGCLR      ( --> )
                  Clears the attn key flag (does not
                  flush attn key from buffer)
WaitForKey      ( --> #KeyCode #Plane )
                  Returns next fully formed keystroke.
```

21.3 InputLine

The word InputLine is the core of the user word INPUT as well as the prompt for equation names (NEW). InputLine does the following:

+ Displays the prompt in display area 2a,

+ Sets the keyboard entry modes,

+ Initializes the edit line,

+ Accepts user input until [ENTER] is either explicitly or implicitly pressed,

+ Parses, evaluates, or just returns the user-input edit line,

+ Returns TRUE if exited by Enter or FALSE if aborted by Attn.

The stack on entry must contain the following:

```
$Prompt    The prompt to be displayed during user input
$EditLine  The initial edit line
CursorPos  The initial edit line cursor position, specified
           as a binary integer character number or a two-
           element list of binary integer row and column
           numbers.  For all numbers, #0 indicates the end
           of the edit line, row, or column.
#Ins/Rep   The initial insert/replace mode:
             #0    current insert/replace mode
             #1    insert mode
             #2    replace mode
#Entry     The initial entry mode:
             #0    current entry mode plus program entry
             #1    program/immediate entry
             #2    program/algebraic entry
#AlphaLock The initial alpha-lock mode:
             #0    current alpha lock mode
             #1    alpha lock enabled
             #2    alpha lock disabled
ILMenu     The initial InputLine menu in the format
           specified by "ParOuterLoop"
#ILMenuRow The initial InputLine menu row number in the
           format specified by "ParOuterLoop"
AttnAbort? A flag specifying whether pressing Attn while
           a non-null edit line exists should abort
           "InputLine" (TRUE) or just clear the edit
           line (FALSE)
#Parse     How to process the resulting edit line:
             #0    Return the edit line as a string
             #1    Return the edit line as a string AND
                   as a parsed object
             #2    Parse and evaluate the edit line
```

InputLine returns different results, depending on the initial value of #Parse:

#Parse	Stack	Description
#0	$EditLine TRUE	Edit line
#1	$EditLine ob TRUE	Edit line and parsed edit line
#2	Ob1 ... Obn TRUE	Resulting object or objects
	FALSE	Attn pressed to abort edit

21.3.1 InputLine_Example

The example call to InputLine shown below prompts the user for a variable name. If the user enters a valid name, the name and TRUE are returned, otherwise FALSE is returned.

```
  ( --> Ob TRUE | FALSE )
::
  "Enter name:" ( *Prompt string* )
  NULL$         ( *No default name* )
  ONEONE        ( *Initial edit line & cursor pos* )
  ONEONE        ( *Insert mode & prog/immed. entry* )
  NULL{}        ( *No edit menu* )
  ONE           ( *Menu row* )
  FALSE         ( *Attn clears edit line* )
  ONE           ( *Return edit line and parsed ob* )
  InputLine     ( ($editline ob TRUE) | (FALSE) )
  NOTcaseFALSE  ( *Exit if Attn pressed* )
  SWAP NULL$?   ( *Exit if blank edit line* )
  casedrop FALSE
  DUPTYPEIDNT?  ( *Check if ob is id* )
  caseTRUE      ( *Yes, exit true* )
  DROPFALSE     ( *No, drop ob and FALSE* )
;
```

21.4 The Parameterized Outer Loop

In this section, the term "parameterized outer loop" is used to refer to a usage of the RPL word "ParOuterLoop", or a combined usage of its fundamental component utilities (described below), all of which can be envisioned as words that take over the keyboard and display until a specified condition is met.

The parameterized outer loop, "ParOuterLoop", takes nine arguments, in order:

- **AppDisplay** The display update object to be evaluated before each key evaluation. "AppDisplay" should handle display updating not handled by the keys themselves, and should also perform special handling of errors.

- **AppKeys** The hard key assignments, a secondary object in the format described below.

- **NonAppKeyOK?** A flag specifying whether the hard keys not assigned by the application should perform their default actions or be canceled.

- **DoStdKeys?** A flag used in conjunction with "NonAppKeyOK?" specifying whether standard key definitions are to be used for non-application keys instead of default key processing.

- **AppMenu** The menu key specification, a secondary or list in the format specified in the menu key assignments document, or FALSE.

- **#AppMenuRow** The initial application menu row number. For most applications, this should be binary integer one.

- **SuspendOK?** A flag specifying whether or not any user command that would create a suspended environment and restart the system outer loop should instead generate an error.

- **ExitCond** An object that evaluates to TRUE when the outer loop is to be exited, or FALSE otherwise. "ExitCond" is evaluated before each application display update and key evaluation.

- **AppError** The error-handling object to be evaluated if an error occurs during the key evaluation part of the parameterized outer loop.

The parameterized outer loop itself returns no results. However, any of the keys used by the application can return results to the data stack or in any manner desired.

21.4.1 The_Parameterized_Outer_Loop_Utilities

The parameterized outer loop word "ParOuterLoop" consists entirely of calls (with proper error handling) to its four RPL utility words, in order:

POLSaveUI Saves the current user interface in a temporary environment. Takes no arguments and returns no results.

POLSetUI Sets the current user interface according to the same parameters required by "ParOuterLoop". Returns no results.

POLKeyUI Displays, reads and evaluates keys, handles errors, and exits according to the user interface specified by "POLSetUI". Takes no arguments and returns no results.

POLRestoreUI Restores the user interface saved by "POLSaveUI" and abandons the temporary environment. Takes no arguments and returns no results.

(In addition to the four utilities above. utility "POLResUI&Err" is used to protect the saved user interface in the event of an error that's not handled within the parameterized outer loop. Refer to "Parameterized Outer Loop Operation" and "Handling Errors with the Utilities", below.)

These utilities can be used by applications that require greater control over the user interface. For example:

+ For optimum performance an application can create a temporary environment with null-named temporary variables after calling "POLSaveUI", then access the null-named variables "within" "POLKeyUI", since only "POLSaveUI" creates a parameterized outer loop temporary environment and only "POLRestoreUI" accesses the same environment.

+ To avoid unnecessary and time-consuming overhead, an application that uses multiple consecutive (not nested) parameterized outer loops can call "POLSaveUI" at the start of the application, then call "POLSetUI" and "POLKeyUI" multiple times throughout the application, then finally call "POLRestoreUI" at the end of the application.

21.4.2 Overview_of_the_Parameterized_Outer_Loop

The parameterized outer loop operates as outlined below.

```
     ("POLSaveUI")
     Save the system or current application's
     user interface

     If error in

       ("POLSetUI")
       Set the new application's user interface

       ("POLKeyUI")
       While "ExitCond" evaluates to FALSE {
         Evaluate "AppDisplay"
         If error in
           Read and evaluate a key
         Then
           Evaluate "AppError"
         }

     Then

       Restore the saved user interface and
       ERRJMP

     ("POLRestoreUI")
     Restore the saved user interface
```

The parameterized outer loop creates one temporary environment when it saves the current user interface, and it abandons this environment when it restores a saved user interface. This means that words that operate on the topmost temporary environment, such as "1GETLAM", should NOT be used "within" the parameterized outer loop (e.g., in a key definition or the application display update object) UNLESS the desired temporary environment is created AFTER calling "POLSaveUI" and abandoned before calling "POLRestoreUI". For temporary environments created before calling the parameterized outer loop, applications should set up and operate on NAMED temporary variables.

21.4.3 Handling_Errors_with_the_Utilities

To insure that it can properly restore a saved user interface if an error occurs within an application, the parameterized outer loop protects the saved user interface by setting an error trap immediately after its call to "POLSaveUI", as shown below:

```
::
  POLSaveUI             ( save the current user interface )
  ERRSET                ( prepare to restore saved user interface
                        in case of error )
    ::
      POLSetUI          ( set the application's user interface )
      POLKeyUI          ( display, read, and evaluate )
    ;
  ERRTRAP               ( if error, then restore the saved
                        user interface and error )
    POLResUI&Err
  POLRestoreUI          ( restore the saved user interface )
;
```

The purpose of supported utility "POLResUI&Err" is to restore the user interface saved by "POLSaveUI" and then to error.

Any applications that use the parameterized outer loop utilities instead of "ParOuterLoop" are REQUIRED to include this same level of error handling protection of the saved user interface.

21.4.4 The_Display

There is no default display in the parameterized outer loop; the application is responsible for setting up the initial display and updating it.

There are two ways that an application can update the display: with outer loop parameter "AppDisplay" or with key assignments. For example, if the user presses the right-arrow key to move a highlight from one matrix column to another, the key assignment for the right-arrow key can either pass information to "AppDisplay" (often implicitly) to handle the change, or the key assignment object can change the display itself. Both methods have advantages under different circumstances.

21.4.5 Error_Handling

The error-handling outer loop parameter "AppError" is responsible for processing any errors generated during key evaluation within the parameterized outer loop. If an error occurs, "AppError" is evaluated. "AppError" should determine the specific error and act accordingly. If an application can not handle any errors, then "AppError" should be specified as "ERRJMP".

21.4.6 Hard_Key_Assignments

Any HP 48 key, in any of the six planes (unshifted, left-shifted, right-shifted, alpha-unshifted, alpha-left-shifted, and alpha-right-shifted) can be assigned for the duration of the parameterized outer loop. The outer loop parameter "AppKeys" specifies the keys to assign and their new assignments.

If a key is not assigned by an application, and outer loop parameter "NonAppKeyOK?" is TRUE, then standard or default key processing occurs, according to outer loop parameter "DoStdKeys?". For example, if user keys mode is on and the key has a user key assignment, then the user key is processed if "DoStdKeys?" is FALSE, or the standard key is processed if "DoStdKeys?" is TRUE. If "NonAppKeyOK?" is FALSE, then all non-application keys issue a canceled key warning beep and do nothing else.

In general, NonAppKeyOK? should be FALSE to maintain total control.

Application key assignments are specified by the secondary object "AppKeys" passed to the parameterized outer loop. The procedure must take as its arguments a key code and a plane specification, and must return the desired key definition and TRUE if the application defines the key, or FALSE if the application doesn't. Specifically, the key assignment procedure's stack diagram must look like this:

```
( #KeyCode #Plane --> KeyDef TRUE )
( #KeyCode #Plane --> FALSE )
```

The key definition result "KeyDef" will be processed by the main key handler, "DoKeyOb".

Application key assignments specified as procedures generally have logic in the form

```
If #Plane is NoShift  (or first plane of interest)
Then
  Process #KeyCode in the unshifted plane
  Else
    If #Plane is LeftShift  (or next plane of interest)
    Then
      Process #KeyCode in the left-shifted plane
  ...
      Else signal no definition
```

This can be implemented in RPL in the form

```
kpNoShift   #=casedrop :: (process noshift plane) ;
kpLeftShift #=casedrop :: (process l-shift plane) ;
2DROP FALSE
```

Each plane handler generally has logic in the form

```
If #KeyCode is 7  (or first key code of interest)
Then
  Return the key code 7 definition and TRUE
Else
  If #KeyCode is 20  (or next key code of interest)
  Then
    Return the key code 20 definition and TRUE
    Else signal no definition
```

This can be implemented in RPL in the following form:

```
kcMathMenu ?CaseKeyDef :: TakeOver (process MTH) ;
kcTan      ?CaseKeyDef :: TakeOver (process TAN) ;
( all other keys )
DROP FALSE
```

In order to save code and to make key definitions more readable, the control structure word "?CaseKeyDef" replaces the

```
     #=casedrop :: ' <KeyDef> TRUE ;
```

portions of code with

```
     ?CaseKeyDef <KeyDef>
```

More specifically, "?CaseKeyDef" is used in the form

```
      ... #KeyCode #TestKeyCode ?CaseKeyDef <KeyDef> ...
```

If "#KeyCode" equals "#TestKeyCode", then "?CaseKeyDef" drops "#KeyCode" and "#TestKeyCode", pushes "KeyDef" and TRUE, and exits the calling secondary. Otherwise, "?CaseKeyDef" drops "#TestKeyCode" only, skips "KeyDef", and continues.

21.4.7 Menu_Key_Assignments

An application can specify any initial menu key assignments, in any of three planes (unshifted, left-shifted, and right-shifted), to be initialized when the parameterized outer loop is started. The outer loop parameter "AppMenu" specifies the initialization object (a list or secondary) for the application's menu, or FALSE, indicating that the current menu is to be left intact. When the parameterized outer loop is exited, the previous menu is restored automatically.

If "AppMenu" is a null list, then a set of six null menu key assignments are made. If "AppMenu" is FALSE, then the menu present when the parameterized outer loop is called is maintained.

NOTE: hard key assignments have priority over menu key assignments. This means that the hard key handler must include the following line if menu keys are to be processed:

```
         DUP#<7 casedrpfls
```

The parameter AppMenu takes the following form:

```
{
  Menu Key 1 Definition
  Menu Key 2 Definition
    ...
  Menu Key n Definition
}
```

Where each menu key definition takes one of three following forms:

```
  NullMenuKey

  { LabelObj :: TakeOver (Action) ; }

  { LabelObj {
               :: TakeOver (Primary Action) ;
               :: TakeOver (LeftShifted Action) ;
             }

  { LabelObj {
               :: TakeOver (Primary Action) ;
               :: TakeOver (LfShifted Action) ;
               :: TakeOver (RtShifted Action) ;
             }
  }
```

A LabelObj may be any object, but is usually a string or an 8x21 grob. See the example below for an illustration of softkey use. The word NullMenuKey inserts a blank menu key which just beeps when pressed.

21.4.8 Preventing_Suspended_Environments

An application may need to allow arbitrary commands and user objects to be evaluated, but don't want the current environment to be suspended by the "HALT" or "PROMPT" commands. If the outer loop parameter "SuspendOK?" is FALSE, then any command that would suspend the environment generates a "HALT not Allowed" error, allowing "AppError" to handle it. If "SuspendOK?" is TRUE, then the application must be prepared to handle the consequences. The dangers here are many and severe.

For all foreseeable applications, "SuspendOK?" should be FALSE.

21.4.9 Specifying_an_Exit_Condition

The outer loop parameter "ExitCond" is an object that evaluates to TRUE when the outer loop is to exited, or FALSE otherwise. "ExitCond" is evaluated before each key evaluation.

21.4.10 ParOuterLoop_Example

```
*---------------------------------------------------------

*
* Include the header file KEYDEFS.H, which defines words
* like kcUpArrow at physical key numbers.
*
INCLUDE KEYDEFS.H
*
* Include the eight characters needed for binary download
*
ASSEMBLE
        NIBASC  /HPHP48-D/
RPL
*
* Begin the secondary
*
::
  RECLAIMDISP           ( *Claim the alpha display* )
  ClrDA1IsStat          ( *Temporarily disable clock* )
*                       ( *Try removing ClrDA1IsStat* )
  ZEROZERO              ( #0 #0 )
  150 150 MAKEGROB      ( #0 #0 150x150grob )
  XYGROBDISP            (  )
*
* Draw diagonal lines.  Remember that LINEON requires
* requires #x2>#x1!
*
  ZEROZERO              ( #x1 #y1 )
  149 149               ( #x1 #y1 #x2 #y2 )
  LINEON                ( *Draw line* )
  ZERO 149              ( #x1 #y1 )
  149 ZERO              ( #x1 #y1 #x2 #y2 )
  LINEON                ( *Draw line* )
*
* Place text
*
  HARDBUFF
  75 50 "SCROLLING"     ( HBgrob 75 150 "SCROLLING" )
  150 CENTER$3x5        ( HBgrob )
  75 100 "EXAMPLE"      ( HBgrob 75 100 "EXAMPLE" )
  150 CENTER$3x5        ( HBgrob )
  DROPFALSE             ( FALSE )
  { LAM Exit } BIND     ( *Bind POL exit flag* )
  ' DispMenu.1          ( *Display Action shows menu* )
  ' ::                  ( *Hard key handler*   )
    kpNoShift #=casedrop
      ::
         DUP#<7 casedrpfls ( *Enable softkeys* )
         kcUpArrow    ?CaseKeyDef
                          :: TakeOver SCROLLUP ;
         kcLeftArrow  ?CaseKeyDef
                          :: TakeOver SCROLLLEFT ;
         kcDownArrow  ?CaseKeyDef
                          :: TakeOver SCROLLDOWN ;
```

```
         kcRightArrow ?CaseKeyDef
                          :: TakeOver SCROLLRIGHT ;
         kcOn         ?CaseKeyDef
                          :: TakeOver
                             TRUE ' LAM Exit STO ;
         kcRightShift   #=casedrpfls
         DROP 'DoBadKeyT
      ;
    2DROP 'DoBadKeyT
  ;
  TrueTrue              ( *Key control flags* )
  {
    { "TOP" :: TakeOver JUMPTOP ; }
    { "BOT" :: TakeOver JUMPBOT ; }
    { "LEFT" :: TakeOver JUMPLEFT ; }
    { "RIGHT" :: TakeOver JUMPRIGHT ; }
    NullMenuKey
    { "QUIT" :: TakeOver TRUE ' LAM Exit STO ; }
  }
  ONEFALSE              ( *1st row, no suspend* )
  ' LAM Exit            ( *App exit condition* )
  ' ERRJMP              ( *Error handler* )
  ParOuterLoop          ( *Run the ParOuterLoop* )
  RECLAIMDISP           ( *Resize and clear display* )
  SetDAsBAD             ( *Redraw display* )
;
```

The above code, if stored in a file SCRSFKY.S, can be compiled as follows:

```
RPLCOMPILE SCRSFKY.S
SASM SCRSFKY.A
SLOAD -H SCRSFKY.M
```

This example also assumes that the file KEYDEFS.H is either in the same directory or the source file has been modified to reflect the location of KEYDEFS.H. The loader control file SCRSFKY.M looks like this:

```
OU SCRSFKY
LL SCRSFKY.LR
SU XR
SE ENTRIES.O
RE SCRSFKY.O
```

The final file, SCRSFKY, may be binary downloaded to the HP 48 for a test.

When SCRSFKY is running, the arrow keys scroll the display, and the labeled softkeys move the window to the corresponding boundary. The [ATTN] key terminates the program.

22. System Commands

The following words set, test, or control various system conditions or modes.

```
ALARM?           ( --> flag )
                   Returns TRUE if an alarm is due
AtUserStack      ( --> )
                   Declares user ownership of all objects
                   on the stack.
CLKTICKS         ( --> hxs )
                   Returns 13 nibble hex string reflecting
                   the number of ticks since 01/01/0000.
                   There are 8192 ticks per second.
ClrSysFlag       ( # --> )
                   Clears system flag from #1 to #64
ClrUserFlag      ( # --> )
                   Clears user flag from #1 to #64
DATE             ( --> %date )
                   Returns real number date
DOBEEP           ( %freq %duration --> )
                   BEEP command
DOBIN            ( --> )
                   Set base mode to BINary
DODEC            ( --> )
                   Set base mode to DECimal
DOENG            ( # --> )
                   Set ENG display with # (0-11) digits
DOFIX            ( # --> )
                   Set FIX display with # (0-11) digits
DOHEX            ( --> )
                   Set base mode to HEXadecimal
DOOCT            ( --> )
                   Set base mode to OCTal
DOSCI            ( # --> )
                   Set SCI display with # (0-11) digits
DOSTD            ( --> )
                   Set STD display mode
DPRADIX?         ( --> flag )
                   Returns TRUE if current radix is .
                   Returns FALSE if current radix is ,
SETDEG           ( --> )
                   Set DEGREES angle mode
SETGRAD          ( --> )
                   Set GRADS angle mode
SETRAD           ( --> )
                   Set RADIANS angle mode
SLOW             ( --> )
                   15msec delay
TOD              ( --> %time )
                   Returns time of day in h.ms form
TestSysFlag      ( # --> flag )
                   Returns TRUE if system flag # is set
TestUserFlag     ( # --> flag )
                   Returns TRUE if user flag # is set
VERYSLOW         ( --> )
                   300 msec delay
VERYVERYSLOW     ( --> )
```

```
                 3 sec delay
WORDSIZE        ( --> # )
                  Returns binary wordsize
dostws          ( # --> )
                  Stores binary wordsize
dowait          ( %seconds --> )
                  Waits for %seconds in light sleep
```