rpl

A Compiled Language for the PET/CBM Series of Computers

A Product of
Samurai Software
P.O. Box 2902
Pompano Beach, FL 33062

Contents

Standard Disclaimer

PART I

GETTING ACQUAINTED WITH RPL

The RPL Compiler is a facility designed to translate programs written in the RPL computer language into a form suitable for execution. Once a given RPL program has been translated by this compiler, it may be executed, or "run", any number of times. There are six main advantages that RPL has over BASIC:

1) The speed of execution of a given RPL program will typically be 6 to 12 times faster than the speed of the equivalent program in BASIC.

2) RPL programs tend to take up somewhat less memory space than their equivalents in BASIC, which means that more space is available for data storage, etc.; this is particularly true for longer programs.

3) RPL programs tend to need far less memory for storage of data than their BASIC equivalents, which again means that they leave more memory available for other uses.

(And, for the advanced programmer:)

4) Structured-programming techniques are easier to apply in RPL due to its "push-down stack" orientation, convenient subroutining capability, nestable "if-then-else" construct, and so on.

5) RPL incorporates a number of desirable features facilitating software development, including: the ability to specify where in memory the program and its data are to reside, the use of "compile-time" symbolic constants, symbolic subroutine names, symbol names of arbitrary length, easy manipulation of pointers and data blocks, etc.

6) The distinction between local and global symbol tables in RPL permits separate compilation of program sections, and facilitates the development of true "subroutine libraries". In addition, the symbol table format used is fully compatible with that used by the Samurai 6502 Assembler and debug packages, permitting both symbolic debugging and cross-referencing of symbols between RPL and assembly language programs.

The disadvantages of RPL relative to BASIC are:

1) A given program in RPL will typically be more difficult to write, and, once written, more difficult to read, than the equivalent program written in BASIC.

2) RPL does not have certain built-in functions that BASIC does, to wit: floating-point arithmetic, the trigonometric

functions, logarithms and exponentials, and IEEE device I/O; these functions may all be accomplished using RPL, but they require some effort, whereas under BASIC they are available via language "primitives".

The question as to whether RPL or BASIC should be used for a given programming project, then, will depend on circumstances. In general, BASIC is better for "quick and dirty" jobs and jobs involving substantial amounts of higher mathematics, whereas RPL is better for sophisticated general program development, particularly where execution speed, memory space, and/or program structure are of concern.

Using RPL on Commodore Computers

Unlike many packages of this type designed for the Commodore product line, the RPL Compiler is designed to interfere as little as possible with the normal operation of the machine. Once the compiler module has been loaded and run, you are left in the normal BASIC command environment, and may continue to use the machine for the most part as though the RPL Compiler were not even there. Then, when you are ready to enter an RPL program, you simply type it in as though it were a new BASIC program, line numbers and all. When you are ready to compile the program, you simply enter the command "compile" and hit RETURN (this will cause the compiler to ask you a couple of questions which we will come to later). Once the program has been compiled, you may run it at any time by simply typing in the word "go" and RETURN. The program may be saved on disk or tape in the usual way at any time through the use of the "save" and/or "dsave" commands, just as though it were a BASIC program. You may of course also load a new BASIC or RPL program at any time through the use of the "load" and/or "dload" commands.

The net effect of all this is that, once you have loaded and run the compiler module, you may load, list, edit, save, and execute both RPL and BASIC programs in any arbitrary sequence, the only difference being that in order to execute a BASIC program you type in "run" and RETURN, whereas to execute an RPL program you type in "compile" and RETURN, answer the two questions it asks (which will almost invariably consist of hitting RETURN twice), and, when it is done, type in "go" and RETURN.

Let's run through a short sample program in both BASIC and RPL to get a feel for this. First, load and run the compiler module, which you do in exactly the same way you would load and run any other program:

```
you type in ----> dload"*"          (or 'load"*"' for cassette)
it comes back --> searching for 0:*
                  loading
                  ready.
you type in ----> run
it comes back --> ready.
```

(You may know of a shortcut for this procedure involving the use of the RUN/STOP key.) Now suppose you want to write a program which will create an interesting visual effect by means of POKE-ing random characters into random positions on the video display. You would enter the following BASIC program:

```
10 poke 32768+rnd(1)*2000,rnd(1)*256
20 goto 10
```

To execute this, you of course just say "run", and to stop it, you hit the RUN/STOP key. Now clear this program out of memory using the "new" command, and enter the following program in its place:

```
10 test: rnd rnd 2000 \ 32768 + poke test goto
```

Now type in the word "compile", hit RETURN, then hit RETURN twice more. The lower portion of your screen should now look like this:

```
10 test: rnd rnd 2000 \ 32768 + poke test goto
compile
rpl compiler (c) 1981 by tim stryker
enter object destination: default
enter gbl sym tbl option: continue
0 errors: code goes from xxxxx ($xxxx) to xxxxx ($xxxx)
ready.
```

The program is now compiled; to execute it, type in "go" and RETURN. Runs a little faster than the BASIC one did, eh? To stop it, hit RUN/STOP. Clear the screen and say "go" again. It does the same thing. Stop it, do a "new", and say "go" again. Same thing again, right? Note that once you have compiled an RPL program you no longer need to keep the "source code" for it around (the source code is the stuff you get when you say "list"). The compiler has translated that source code into "object code", which is stored elsewhere in memory (more about that later). Keep in mind, though, that when you do a "save" or a "dsave", you are only saving the source code. For now, in order to execute the program again after having lost the object code (either by turning the machine off or by compiling something else over it), you will have to re-enter or reload the source from tape or disk, and then re-compile it. Incidentally, note that loading and executing other BASIC programs will not ordinarily destroy the object code created during your last compilation. For example, you could now

load that first BASIC program for creating the visual display back into memory, and switch back and forth between executing the BASIC version and executing the RPL version by alternately issuing "run" and "go" commands.

The RPL Language

RPL stands for "Reverse Polish Language". It is patterned after the popular computer language FORTH, which also uses "Reverse Polish Notation". RPL, however, embodies a number of advantages over FORTH, among them more efficient use of memory and a much friendlier user interface. For those who are interested, a discussion of the main differences between RPL and FORTH is given in Appendix A.

The term "Reverse Polish Notation" refers to a method of specifying the order in which mathematical operations are to be performed, pioneered by the Polish mathematician Jan Lukasiewicz earlier this century. If you have ever used a Hewlett-Packard calculator, you have seen how this works. Suppose you want to print out the result of adding 5 to 3. In BASIC, you would of course simply write "print 5+3". Things are a little more complicated in RPL, but with good reason: bear with me here.

In RPL there is something called a "parameter stack", which can be thought of as one of those tubes of lunch plates you see in cafeterias all the time -- you know, the kind with the spring in the bottom, so that when you put a plate on the top of the stack, the rest move down a notch, and when you pull it back off, the rest move back up. The idea behind the parameter stack is that if you want to perform some operation on two numbers, you first put the two numbers onto the stack, one after the other, and you then say "plus!" or "minus!" or whatever the operation may be. What happens then is that the two operands are "popped", or removed, from the stack, and their sum or difference or whatever is then pushed back onto the stack. Thus, in RPL, to add 5 to 3, you would specify that you want to push a 5, then push a 3, then add, which you do by saying "5,3,+". When executed, this little three-step "program" will leave an 8 sitting on top of the stack.

Now, having an 8 sitting on top of the stack is all very fine, but in order to get this result printed out on the video display, you have to do a little more. Specifically, you have to convert this result into printable form, and then you have to actually ask for the result of that conversion to be printed out. To do this, you say "str$", which pops the number on top of the stack and then pushes a character-string equivalent of it back on, after which you say "print", which invokes a little program who expects to

find a character-string on the stack, and who pops it off as he prints it out. So: the RPL equivalent of BASIC's "print 5+3" is "5,3,+,str$,print". One way to read this is: "push a 5, push a 3, add them together, convert the result into ASCII, and print it out". In order to turn this into a real live program, you would also have to place a line number at the start of the line. So, the full program line which you could actually type in, compile, and execute, if you wanted to, would be:

```
10 5,3,+,str$,print
```

(PET/CBM afficionados will undoubtedly gag at the sight of this line, thinking that the space between the 10 and the 5 will be ignored, leading to a line 105 which goes ",3,+,str$,print". Have no fear. The compiler module includes some code that eliminates this type of problem.)

Table 1 contains all of the valid RPL operators and a brief description of what each one does. We will discuss each one in some detail later, but we should go over the question as to how data is represented in RPL first.

On the parameter stack, each piece of numeric data is represented using sixteen base-2 digits, i.e., two bytes. This means that integers ranging from -32768 up to 32767 can be conveniently dealt with (this range may also be viewed as extending from 0 to 65535, if desired, as long as you watch out for a few things we will come to later). When negative, these numbers are represented using "2's complement" notation. See Appendix B for an explanation of 2's complement if you are not familiar with it -- it is basically a very simple scheme whereby a negative one is represented by a bit pattern of all ones, a negative two is represented by fifteen ones followed by a zero, etc.

Character strings, which can contain any number of characters, are represented a little differently. Each of the characters in a string is allocated its own entry in the stack, and at the very front of the string (in the topward direction on the stack), the number of bytes in the string appears. For example, if you were to push the string "hello" onto the stack, you would find that the topmost stack entry is a 5 (the string length), the next item down is a 72 (the CBM code for the letter "h"), the next item down is a 69 (CBM code for "e"), and so on until the sixth item down is a 79 (CBM code for "o"). Once a character string has been pushed onto the stack, or has been created there by some operator like "str$", it need not be immediately printed out: if you want to you may manipulate the pieces of it using other ordinary stack operators, just as you would any sequence of stack entries. In this way you may mimic the BASIC operations of "left$", "right$", string concatenation, and the like. Conversely, there is nothing to

TABLE 1: The RPL Operators

(TOS means "Top Of Stack"; NOS means "Next On Stack")

Operator	Description
+	Replace NOS with NOS plus TOS, pop TOS
-	Replace NOS with NOS minus TOS, pop TOS
*	Replace NOS with NOS times TOS, pop TOS
/	Replace NOS with NOS divided by TOS, pop TOS
\	Replace NOS with NOS modulo TOS, pop TOS
str$	Convert TOS to a character string in base ten
chr$	Convert TOS to a character string in hexadecimal
"..."	Push character string as defined within quotes
print	Output the character string on TOS to video display
>	Replace NOS with logical NOS > TOS, pop TOS
<	Replace NOS with logical NOS < TOS, pop TOS
=	Replace NOS with logical NOS = TOS, pop TOS
and	Replace NOS with NOS Boolean-AND TOS, pop TOS
or	Replace NOS with NOS Boolean-OR TOS, pop TOS
not	Replace TOS with its 1's complement (invert bits in TOS)
#	Push another copy of TOS ("duplicate")
;	Push another copy of NOS ("over")
↑	Replace TOS with the TOS-th deep stack entry ("N-th")
%	Swap NOS with TOS ("swap")
$	Rotate TOS-th stack entry out to TOS ("rotate")
.	Pop TOS ("drop")
new	Pop everything on parameter stack
int	Interchange high- and low-order bytes in TOS
@	Replace TOS with 2-byte word that TOS points to
!	Store 2-byte NOS word at address on TOS, pop TOS and NOS
peek	Replace TOS with single byte that TOS points to
poke	Store single byte NOS at address on TOS, pop TOS and NOS
&	Call RPL routine whose address is on TOS, pop TOS
return	Return to caller
sys	Call 6502-language routine whose addr is on TOS, pop TOS
goto	Go to address on TOS, pop TOS
if	If TOS = 0, go to corresponding "then" or "end", pop TOS
then	Begin definition of else-clause for corresponding "if"
end	Define end of if-clause or else-clause
for	Start for/next loop, from TOS to NOS, pop TOS and NOS
next	End for/next loop
fn	Push current for/next loop counter value
clr	Clear out current for/next loop context
rnd	Push a random 2-byte quantity
get	Push a byte from the keyboard, or 0 if none available
input	Await input from keyboard; push character string result
stop	Halt RPL execution, return to BASIC

prevent you from building your own strings from scratch if you like: the program "10 79,76,76,69,72,5,print" will print out "hello" like nobody's business. Try it.

Let's take a look at each of the classes of RPL operators now, in detail.

The Arithmetic Operators: +, -, *, /, and \

These operators are all similar in that they combine the top two items on the stack in some way, pop them both, and then push a result. The order of the input operands, where it matters, is always given by the rule: NOS operator TOS. For example, to divide 2200 by 3, write "2200,3,/"; to obtain the remainder from that division, 2200 modulo 3, write "2200,3,\".

For the addition, subtraction, and multiplication operators, it makes no difference whether numbers with their high-order bits set are viewed as being in the range from -32768 to -1 or in the range from 32768 to 65535. For division and modulo, however, it does make a difference: the sequence "65535,5,/" will yield 0, not 13107. If you expect a dividend in a division or modulo operation to exceed 32767, it would be best to do the operation in stages. The result of a modulo operation, incidentally, will always be the remainder left from the division of the absolute value of the dividend by the absolute value of the divisor.

The Numeric-to-String Conversion Operators: str$ and chr$

Both of these operators pop the top stack entry and then push a character string equivalent of it back on. Str$ is similar to its namesake in BASIC in that the string it returns is in ASCII decimal (base ten). The string is returned with no leading or trailing spaces of any kind, and will appear as a negative number if the high-order bit of the input argument is set. Chr$, on the other hand, is nothing like its BASIC namesake: it returns a four-character string representing the top stack entry in hexadecimal notation (base sixteen). In case you are not familiar with this notation, Appendix B contains a brief explanation of it. Note, incidentally, that the true RPL equivalent of BASIC's chr$ function consists of nothing more than pushing the number 1: this will convert a number into a one-character string suitable for printing.

The String-Push Operation

It is frequently useful to be able to simply push a literal

character string directly onto the stack for use in labelling output, prompting for input, etc. This is done by enclosing the string in double-quotes: just as in the case of the numeric push, by simply calling out the entity in question, you are understood to be asking for it to be pushed onto the stack. The only limitation on strings pushed in this way is that they may not contain the double-quote symbol. If you need to push this symbol you will have to push it in numerical form (CBM code 34).

The Output-to-Video Operator: print

This is one we have seen before. It looks at the top stack entry to determine how many characters to print out, then pulls that many characters off the stack, one by one, and outputs them to the screen. It does not output a carriage-return automatically when it is done: if you want a RETURN output, incorporate an ASCII 13 into your string before outputting. "Print" only considers the low-order byte of each of the stack entries composing the string, in case that makes any difference. Also, in case you are wondering, it will handle the null string, a zero on TOS with nothing after it, by simply popping the zero and not printing anything. In any event, it always pops the entire input character string before returning.

The Comparison Operators: >, <, and =

These operators are most commonly used in conjunction with the "if" operator, although, as in BASIC, their results may also be used in computations. What they do is to apply the specified relation to the top two stack entries, popping those entries, and pushing in their place a -1 (bit pattern of all ones) if the relation holds true between them; otherwise pushing a zero. Where order matters, the rule is always: NOS operator TOS; for example, "4,7,>" will yield a zero, whereas "4,7,<" will yield a -1.

The > and < operators will consider any number with its high-order bit set to be negative, so if you wish to treat these numbers as being in the range from 32768 to 65535, you will have to watch out for this. For example, the sequence "60000,3,>" will yield a zero. See the "Getting It All Together" section of this manual for an example of an unsigned compare routine.

The Boolean Operators: and, or, and not

The "and" operator performs a bitwise Boolean AND of the top two stack entries, pops them, and pushes the result. This means that in any bit position in which both TOS and NOS have a 1, the result

will have a 1; all other bit positions in the result will contain 0. This is primarily useful for combining results of comparisons; for example, suppose we need to form a logical quantity (either -1 or 0, signifying true or false) which is true if and only if the quantity now on the stack is both greater than 47 and less than 58: we could write "# 47 > ; 58 < and" (see below for explanations of the "#" and ";" operators).

Boolean OR works much the same way, except that when the TOS and NOS are OR'ed together, each bit position in which either TOS or NOS (or both) has a 1 will yield a 1 in the corresponding bit position in the result. Only bit positions in which both TOS and NOS contain zeroes will produce bit positions containing zeroes in the result.

The "not" operator simply complements, or inverts, each bit in the TOS. This is primarily useful for inversion of logical quantities, such as the result of a comparison for equality.

The Stack-Replicating Operators: #, ;, and ↑

All three of these operators pick up an item from somewhere in the stack and make a new copy of it on top of the stack. "#" (pronounced "dupe", short for "duplicate") picks that item up from the existing TOS itself: the sequence "14,#" yields two 14's, the original one being now the NOS, the new one being the TOS. ";" (pronounced "over") picks up the NOS and pushes a new copy of it: the sequence "9,5,;" will yield a 9 as TOS, a 5 as NOS, and the original 9 as the third item deep in the stack. Both "#" and ";" are simply shorthand forms of the general operator "↑" (pronounced "n-th"), which uses the TOS as an index into the stack to determine which item there to pick up. The sequence "1,↑" is equivalent to "#"; "2,↑" is equivalent to ";". To get a fresh working copy of the next item down past NOS, you need simply say "3,↑", and so on.

The Stack-Shuffling Operators: % and $

From time to time you will reach a point at which you need to interchange the top two stack entries in preparation for a subtraction, a division, or some other purpose: the "%" (pronounced "swap") operator will accomplish this for you. Occasionally, you will also wish there were some way to effectively do an n-th and at the same time collapse the stack to remove the item accessed -- to, in a sense, "rotate" a given item, perhaps deep in the stack, up to the TOS position. Well, there is, and the "$" (pronounced "rotate") operator is the key to it. It is used much like the "↑" operator: the sequence "2,$" is equivalent to "%";

to rotate out the third item deep, you say "3,$", and so on.

The Poppers: . and new

It is frequently useful to be able to simply pop an item off of the TOS without doing anything further to it. This is accomplished by saying simply "." (pronounced "drop", or "pop", as you please).

Somewhat less frequently you may find that you need a way to clear off the entire parameter stack, without knowing exactly how many items there are in it at the time. The "new" operator will do this for you.

The Byte-Interchanger: int

The 6502 microprocessor, which the PET/CBM series uses, has a peculiar standard method for dealing with double-byte data. When stored in memory, a double-byte quantity is almost always stored with the low-order byte of the quantity in the lower of the two addresses. RPL conforms to this standard as well, but there are occasions when the opposite order is desired. In order to deal with these occasions, you will need the "int" operator, which simply interchanges the two bytes constituting the TOS. For example, the sequence "513,int" will yield a 258: the 513 consists of two bytes, the high-order byte containing a 2 and the low-order byte containing a 1; by interchanging these we get a high-order byte of 1 and a low-order byte of 2, which is 258 in decimal.

The Memory-Access Operators: @, !, peek, and poke

These operators constitute the primary means whereby information is transferred back and forth between the stack and main memory. "@" treats the TOS as a pointer to a 2-byte quantity in memory (stored low-order byte first), and replaces that pointer with the data it points to. "!" (pronounced "store") takes the NOS and stores it into the two bytes pointed to by the TOS, popping them both in the process. The low-order byte is stored into the lower of the two addresses, as you would expect.

Peek and poke work identically to "@" and "!", except that only the low-order byte of data is involved. Peek fetches only one byte from memory, filling the high-order byte of the result on the stack with zeroes, while poke stores away only one byte, ignoring the high-order byte of the NOS.

Ordinarily, the address operands to all four of these operators

will be referenced symbolically in your source code: symbol-handling techniques will be covered at the end of this section.

The Subroutine Operators: & and return

The key to "structured programming" is a powerful and convenient subroutining capability. RPL supplies this capability in the following form: the address of the routine to be called is brought to the TOS, and the "&" (pronounced "call") operator is invoked. "&" pops the TOS and uses it as an address to transfer control to. It also places the address of the "&" itself on a new stack we haven't talked about yet, called the "return stack". The return stack is much less accessible to you than the parameter stack is, and for the most part you need not even be aware it exists. However, it is needed so that the next time a "return" operator is executed, control can pick back up at the point following the last call. Rather than giving a example here of the use of a subroutine, let's wait until we have discussed the uses of symbols, which are indispensable to this purpose.

Parameters may be passed to the called routine either on the parameter stack or through memory; to pass them on the stack you simply put them there, put the routine address on top of them, and say "&": when the called routine gets control its own address will already have been popped, so the parameters it needs will be sitting right there on top of the stack. Return codes of any kind may be passed back to the caller in the same general way, either on the stack or through memory. It is completely up to you whether or not a routine pops the parameters passed to it before returning -- in general, it is a good idea for routines to do so, but there exist circumstances in which a routine will want to leave the passed parameters alone. In any event, neither "&" nor "return" has any effect whatsoever on parameters passed in either direction, so you will have to keep track of any passed parameters yourself.

The Machine-Language Access Operator: sys

This operator is provided for the use of advanced programmers for whom not even RPL is fast enough. Unless you have some knowledge about assembly language and machine language on the 6502, you will not need this operator (see Programming the 6502, by Rodnay Zaks, or any of a number of similar books for more information on this fascinating subject).

Like BASIC's "sys" command, this operator executes as a JSR to the address specified. The address is taken from the TOS, and is

popped. Upon entry to the target routine, the Y register may be assumed to contain 0, and the X register to contain a copy of the stack pointer, as though a TSX instruction had already been done for you by the time you get control. The RPL parameter stack is the hardware stack. Thus, for example, the instruction "LDA $103,X" will pick up the low-order byte of what had been the TOS before you pushed the routine address and did the sys. If you are planning to pass parameters back and forth on the stack between RPL and machine language, keep in mind that your routine's own return address will be foremost on the stack at the time it is called.

Machine language routines may be constructed inline, semi symbolically, through the use of parentheses and brackets (see the routine "fndsym" in Appendix G for an example). If you are thinking of writing substantial amounts of assembly code, though, you would be well-advised to purchase the Samurai 6502 Assembler, which includes all the usual goodies like precedence parsing of expressions, macros, and conditional assembly, plus the same local and global symbol table architecture as the RPL compiler. This means that your RPL code and your assembly code can share symbols in both directions; memory allocations may automatically be made contiguous and non-overlapping, etc.

See Appendix E for information concerning memory locations reserved for use by the RPL run-time package (and be careful not to clobber them!).

The Unconditional-Control-Transfer Operator: goto

The bugaboo of structured programming, "goto" is nonetheless useful in many situations. To use it, you simply place the address to transfer control to (almost invariably referenced symbolically) on the stack and say "goto". The address itself will be popped, and execution of RPL code will pick up starting with the byte at the specified address.

The Conditional Clause Operators: if, then, and end

Any reasonable computer language needs a method of saying "if such-and-such is the case, then do this, otherwise do that". In RPL, the way you do this is by bringing to the TOS a quantity that has the potential to be either zero or nonzero. Then, by invoking the "if" operator, you are specifying that you wish the subsequent code to be executed only if that quantity on TOS is nonzero. You have two things you may do from that point on: you may place the "end" operator at the end of the section of code you want to make conditional, in which case, if the argument to the "if" is zero,

control will simply branch to that "end"; or, you may form an "else-clause" by invoking the "then" operator, writing the clause, and placing the "end" at its finishing point. For example, suppose you wanted to print out either the word "zero" or the word "nonzero", depending on whether or not the quantity at TOS is or is not zero. One way to do this would be to write:

```
if,"nonzero",print,then,"zero",print,end
```

A shorter way to achieve the same thing would be to say:

```
if,"nonzero",then,"zero",end,print
```

And the shortest way of all would be:

```
if,"non",print,end,"zero",print
```

RPL permits full nesting of conditional clauses (this means you may have, say, one if-clause within another if's else-clause, which itself is... and so on). Program line boundaries have no effect on the scope of an "if", so you may construct arbitrarily complex conditional clause structures, up to a maximum nesting depth of 38, which should be sufficient for most purposes. The only requirement is that every "if" must have a corresponding "end", and vice versa; whether or not each one also has a "then" is up to you.

It should be noted in passing here that "end" is not really an operator -- it does not get translated into any actual object code. It is what is called a "pseudo-op": a token that tells the compiler something about what to do -- in this case, where to cause control to branch to when the real operators "if" and "then" are executed. But once your program has been compiled, any "end"s you have in your code will take up no space, and will consume no time during execution.

The Counting-Loop Operators: for, next, fn, and clr

It is frequently useful to be able to set up a section of code that will be repeatedly executed for a specified number of times, the way one does in BASIC using a "for/next loop". RPL provides a similar capability: by pushing the final value for the loop counter, followed by its initial value, and then saying "for", a loop will be set up such that the next time that the "next" operator is encountered, a loop counter will be incremented and control will pass back to the point immediately following the "for"; this will continue to happen each time "next" is encountered, until the loop counter reaches the final value specified. For example, the sequence "5,1,for,7,str$,print,next"

will print out five 7's on the video display. In this example the final value (or upper bound) for the loop counter is specified as 5, and the initial value (lower bound) is a 1, so the "body" of the loop, the sequence "7,str$,print", will be executed five times. The fifth time through, when the "next" is reached, the loop will be "exited", and control will pass to the code following the "next", whatever that is.

In the example above, no use was made of the actual value of the loop counter. Suppose you were to want to achieve the effect of the BASIC line "for i=-5 to -2:print i;:next". The RPL operator "fn" causes the present value of the "innermost" loop counter active to be pushed onto the stack, so in RPL you would write "-2,-5,for,fn,str$,print,next". Loops may be nested (i.e., you may have one for-next loop inside another, and another inside that, if you like, etc.), so keep in mind that it is the value of the innermost loop counter that is pushed by "fn". Loop counters of outer loops are unavailable, unless you have had the foresight to push a copy of their loop counters before entering the innermost loop. See the "Getting It All Together" section of this manual for an example of this.

It is sometimes useful to be able to exit a for/next-type loop without completely exhausting the range of the loop counter. For example, in BASIC, to search program memory for the first byte containing a 37 you might say:

```
10 for i=1024 to 32767:if peek(i)<>37 then next
```

When this line finishes executing, i is equal to the address of the first byte containing 37 or to 32768, whichever came first. Moreover, if the search does find a 37, there is no requirement for your program to "finish out" the loop, running i through the rest of its range up to 32767. BASIC has a very complex internal procedure for handling this kind of situation, and in fact this is one of the sorts of things that makes BASIC so slow. To keep things fast, in RPL you have to explicitly specify that the loop will not be completed, in the event that this is the case. This is not difficult: all you need to do is to invoke the "clr" operator once you have decided not to "finish out" the loop. The RPL equivalent of the above line of BASIC would thus be:

```
10 32767,1024,for,fn,peek,37,=,if,fn,clr,then,next,32768,end
```

This program segment will leave either the address of the first 37, or 32768, whichever came first, sitting on TOS, and will leave the return stack, which is where for/next context is stored, in pristine condition. (Incidentally, in case this example raises doubts in your mind concerning the claim that the use of RPL yields code that is smaller as well as faster than BASIC's,

consider that the RPL object code in this case is 21 bytes long, whereas its BASIC equivalent takes up 30 bytes, not counting spaces or storage for the variable i. See Appendix D.)

The Random-Number-Generator Operator: rnd

This operator does nothing more than push a random 16-bit quantity onto the stack. Each time it is invoked, a new random number will be pushed -- however, you should be aware that, viewed as bit patterns, successive random numbers are not wholly independent of one another: the algorithm used is of the shift-and-XOR class, if that means anything to you. At any rate, the sequence will not begin repeating itself until the 16,777,215th time around, and when used in conjunction with the modulo function, as it almost always will be, the numbers derived from it should appear totally random.

The Keyboard-Input Operators: get and input

These two operators resemble their namesakes in BASIC very closely. "Get" checks the keyboard input buffer, and, if anything is there, pushes the first character in it onto the stack; if the buffer is empty, it pushes a zero. Note that it does not push an RPL-style character string (with length entry of 1), but merely pushes the character itself, or, more precisely, its CBM code. "Input" on the other hand puts the flashing cursor up on the screen and waits for the user to type something in and hit a carriage return, just the way BASIC's "input" does (RPL's "input", however, does not automatically print out a question mark). It then takes the entire resulting string and pushes it onto the stack. Note: due to the maximum stack capacity of 63 entries, if an input string is over 62 characters long, stack overflow is guaranteed. For applications in which this is a problem, you may want to consider writing your own input routine using "get".

The Execution-Termination Operator: stop

This operator does exactly what you would expect: it terminates execution of your RPL program and returns you to the BASIC command environment. As is the case in BASIC, execution will also terminate if control "falls off the end" of your code. (Be careful when making use of this feature, though: if your program allocates any memory for data storage, you must ensure that execution will not pass through this data area on its way to "falling off the end" of the program. This will become clearer below in the section on symbols.)

Once RPL execution is terminated via "stop", it may not be restarted again with the "cont" command, the way it can in BASIC. See the "Associated Products" section at the end of this manual for a description of the Samurai RPL Symbolic Debugger, which gives you this capability, and can assist you in many other ways as well.

Symbols

Consider the following BASIC program:

```
100 q$ = "do you want instructions? "
200 gosub 1000
300 if y then print"well, there aren't any."
400 q$ = "is your name fred? "
500 gosub 1000
600 if not y then print"then go away."
700 end
1000 print q$;
1100 get a$ : if a$="" then 1100
1200 print a$
1300 if a$="y" then y = -1 : return
1400 if a$="n" then y = 0 : return
1500 print"please hit either 'y' or 'n': ";
1600 goto 1100
```

Here we have a classic example of the use of a subroutine: the portion of the program from line 1000 to line 1600 would have to be written out twice, once between lines 100 and 300, and once between lines 400 and 600, if the BASIC "gosub" command were not used. In BASIC whenever you want to transfer control to another portion of code, using either "gosub" or "goto", you always refer to that portion of code by line number.

In RPL, you do not use line numbers. Instead, you label each place in your code that gets branched to from someplace else with a symbol, and then you refer to those places by means of the symbol name. The way that you define a symbol is by stating the symbol name, following it immediately with a colon. The RPL equivalent of the above BASIC program would go as follows:

```
100 "do you want instructions? "
200 getanswer &
300 if "well, there aren't any." print 13 1 print end
400 "is your name fred? "
500 getanswer &
600 not if "then go away." print 13 1 print end
700 stop
1000 getanswer: print
1100 waitforanswer: get # 0 = if . waitforanswer goto end
1200 13 ; 2 print
1300 # "y" . = if . -1 return end
1400 # "n" . = if . 0 return end
1500 . "please hit either 'y' or 'n': " print
1600 waitforanswer goto
```

This may look a little strange, but take it one step at a time. Note that in lines 200 and 500, instead of saying "gosub 1000", we are pushing the value of the symbol defined in line 1000, "getanswer", and then invoking the "&" (call) operator. Similarly in line 1600 instead of saying "goto 1100", we push the value of "waitforanswer" and then say "goto". A similar effect applies in line 1100 of both versions.

This example also gives us a chance to see parameter passing in action. In the BASIC version, q$ was used to pass the prompt string to the subroutine, and y was used to return the indication as to whether the answer was yes or no. In the RPL version both of these functions are carried out directly on the stack. The string-pushes in lines 100 and 400 of the RPL version get passed to getanswer, which immediately prints them out; in lines 1300 and 1400 the numeric pushes occurring just prior to the "return"s are what the "if" and "not" operators in lines 300 and 600 act upon. The rest of the RPL version is pretty straightforward: all of the #-ing and .-ing you see is just stack management, and will become second nature to you after a while. Refer to Table 1 and the operator descriptions if you have any trouble.

The other use of symbols for which a parallel exists in BASIC is that involving variables. So far none of our examples has used main memory for variable storage -- all of our data has been kept on the stack. It is generally only necessary to store data off the stack when you start getting into more complex applications, so the following example may seem contrived. Take this BASIC program:

```
10 get a$ : if a$="" then 10
20 x = x+1
30 print x
40 goto 10
```

Every time you hit a key, this program spits out a count of how many keys you have hit so far. Here is one way to write an RPL equivalent:

```
10 awaitkey: get 0 = if awaitkey goto end
20 x @ 1 + x !
30 x @ str$ print 13 1 print
40 awaitkey goto
50 x: [0]
```

The first thing to notice about this is that there is an extra line in the RPL version, line 50. Unlike BASIC, RPL does not automatically allocate space for variables for you, so you have to do it yourself. In fact, we can go further than that: the RPL compiler has no idea of even what a "variable" is, in the sense

used in BASIC; instead, it allows you to allocate memory space, and to apply labels to the space allocated. But it couldn't care less what you use the space for -- you could use it for variables, you could use it for pointers to other variables, you could put programs in there and then branch to them, anything at all. The way in which the symbol "x" in line 60 becomes associated with that particular spot in the program is exactly identical to the way in which the symbol "awaitkey" gets associated with the spot it is in in line 20. The only difference is in how they are used.

By saying "x: [0]" in line 60, what we are doing is allocating two bytes worth of memory space, putting a zero into each byte, and labelling the whole thing "x". Then, in line 20, when we say "x @", what we are saying is, "push the value of the symbol 'x', but instead of branching there, fetch the contents of the memory location pointed to by this symbol, and put those contents on the stack". Once this has taken place, by saying "1 +" we are of course incrementing the value on the stack, which happens to be what was stored at "x". We then say "x !", which means, "push the value of the symbol 'x' again, only this time, take what is sitting on TOS and store it into the 2 bytes pointed to by the symbol 'x'". The net effect of the whole sequence in line 20, then, is that the contents of the 2 bytes stored in the location labelled "x" are incremented: in a sense, this thing we are thinking of as the "variable 'x'", is incremented, just as it is in line 20 of the BASIC version.

In line 30 of the RPL version, we once again fetch the value stored at "x" onto the stack, in order to convert it into a decimal string and print it out (the "13 1 print" in line 30 simply prints a carriage return, in case you were wondering).

Now, this example brings out two important points relating to RPL in general. The first is that you have to draw a distinction between things that happen at "compile-time", and things that happen at "run-time". The compiler is the entity that looks at your source program and sees the "[0]" in line 60: when it sees that, it sticks the two zeroes in those locations, like we discussed, and then goes about its business. Then, when you run the program the first time, and you hit a key that is picked up by the "get" in line 10, the program looks at x, finds a zero there, increments it, and stores it back, etc. Now suppose you run the program a second time without compiling the program again in between. Nothing has come in to set the contents of x back to 0 again, so x will begin incrementing from wherever it left off the first time around. So: unlike BASIC, which zeroes out your variables for you each time you say "run", RPL does only what you specifically tell it to do in the way of storing values into labelled locations. To make the RPL version of our little program here a true equivalent of the BASIC version, you would have to add

a line, say, 5, saying "0 x !", i.e. "store a 0 into the byte-pair labelled 'x'".

The other important point this example brings out is that there is a price you pay for the power that RPL gives you in being able to treat program symbols and data symbols completely interchangably. The price is that if you were to make a mistake in your program, and, for example, say "x goto" at some point, RPL would blithely follow along and begin doing all sorts of things that you had no intention of. Depending on the circumstances, you could easily end up "crashing" your machine, which means that you would lose control of it, and would have to turn it off and back on in order to regain control. Naturally, this means that any programs you had had in memory at the time would be lost, unless you had previously stored them on tape or disk. The moral of this story, then, is: always make a backup copy of any sizable program you type in before running it the first time, just in case you have made a mistake along these lines. "A minute of prevention is worth an hour of cure."

One more minor point on symbols: they must be composed entirely of alphabetic characters. Unlike BASIC, FORTRAN, and the like, RPL will misinterpret symbol names like "c8" and "wazoo22". There is a good reason for this, which we will come to presently. In the meantime, do not try to get RPL to accept symbol names containing numbers.

Using the Square Brackets

In the preceding section, you saw how to allocate two bytes for a variable by following a symbol definition with "[0]". This is a special case of a very general and useful RPL feature that, unfortunately, doesn't really have a name. The way it works is that you may place any arbitrary list of numbers and symbol names within brackets, and what happens is that the compiler will generate that list for you in memory in simple numeric form, two bytes per entry, resolving the symbols that appear, if there are any. In effect this allows you to create predefined arrays of various kinds with very little effort.

For example, suppose you needed a five-element array containing the prices of five different brands of cement mixers for some reason. In BASIC you would probably set this up as follows:

```
10 for i = 1 to 5 : read cm(i) : next
2000 data 5000,6500,5995,5800,6350
```

In RPL all you would need to do is say:

```
2000 cmixprices: [5000,6500,5995,5800,6350]
```

This sets up a sequence of ten bytes in the object code (be careful your program doesn't try to execute this sequence!), the first two containing the number 5000, the next two containing 6500, and so on. To access these, all you need to do is to get the index of the item you want, a number from 1 to 5, on the stack, and say "1,-,#,+,cmixprices,+,@". The "1,-" decrements your index to make it a number from 0 to 4, "#,+" doubles it to make it a byte offset into the price table, "cmixprices,+" adds that offset to the address of the first byte in the array, and "@" then fetches the appropriate array entry.

Another good example of the use of brackets involves the RPL equivalent of BASIC's "on...goto" and "on...gosub" commands. Take the following BASIC subroutine for generating a random insult:

```
1000 on int(rnd(1)*4)+1 goto 1010,1020,1030,1040
1010 print"you bohemian!"; : return
1020 print"take a long walk, off..."; : return
1030 print"drop dead!"; : return
1040 print"what a banana!"; : return
```

One RPL equivalent would be:

```
1000 insult: rnd 3 and # + insulttable + @ goto
1005 insulttable: [insult_a,insult_b,insult_c,insult_d]
1010 insult_a: "you bohemian!" print return
1020 insult_b: "take a long walk, off..." print return
1030 insult_c: "drop dead!" print return
1040 insult_d: "what a banana!" print return
```

Here, instead of looking up an ordinary numeric quantity in an array for purposes of arithmetic computation, we are looking up an address. The application may be completely different, but the underlying concept is exactly the same. (In case you are interested, the underscore character, "_", appearing in the symbols above is what most printers generate when they encounter the PET/CBM back-arrow. RPL treats the back-arrow/underscore character as alphabetic, so that those who like to make long, descriptive labels using this character can do so. Maximum symbol name length is virtually unlimited, and all characters in a name are treated as significant. Practically speaking, lengths in excess of 50 characters or so should be avoided, as they may overflow the compiler's stack).

Commenting RPL Programs

Any reasonable computer language has to allow its user to place

comments (what BASIC calls "remarks") inline. The way you do this in RPL is: you write one program line containing nothing but the word "rem", then you write your comment in the following line or lines, then you write another line also containing nothing but the word "rem". Comments may be placed between any arbitrary pair of lines: since comments take up zero memory space in the object code, execution will pass through them as though they were not there.

There is a neat little feature you may find handy here: if you want to put a block comment at the beginning of your program, you can do so without using a leading "rem". This is due to the fact that the compiler actually compiles your program from bottom to top. What you think of as the beginning of your program is the last thing it looks at, so if it happens to be "reading comments" (or BASIC programs!) when it reaches the top of your code, what's the difference? See below for examples.

Error Handling

A listing of all of the types of errors that the compiler detects, along with an explanation of each, is found in Appendix C. These are all errors that are detectable at "compile-time". It is generally not advisable to issue the "go" command following a compilation in which you have gotten one or more of these error messages.

"Run-time" error checking has been almost completely eliminated from the RPL package in order to keep the speed high and the memory space requirements low. The only checks that are made during execution are for parameter stack overflow and underflow. The stack will overflow whenever you try to put more than 63 items on it, and it will underflow if you try to pop or otherwise make use of a stack item that isn't there. Both of these conditions halt execution immediately, and give rise to the stack error indication "p!".

See the discussion of the Samurai RPL Symbolic Debugger in the Associated Products section at the back of this manual for more information concerning run-time error checking.

Getting It All Together

You should now have enough information to get started writing your own professional-quality RPL programs. Below are a few examples covering applications in which the various features discussed above work in concert with one another.

Example 1: Two Forms of an Unsigned-Compare Routine

```
500 rem
510 this routine treats the two top stack entries as
520   unsigned quantities, pops them, and returns a -1
530   on the stack if the one in the "nos" position
540   was greater than the one in the "tos" position;
550   otherwise it returns a 0.
560 rem
570 usgt: # 0 < if ; 0 < if > then < end
580             then ; 0 < if < then > end end return
600 rem
610 alternate, smaller form of the above
620 rem
630 smusgt: ; 32768 and ; 32768 and = if > return end < return
```

Example 2: A Numeric-Input Routine

```
10000 rem
10010 this routine expects a prompting string to
10020   be passed to it on the stack; it prints
10030   out the string, waits for keyboard input,
10040   and checks to see if the user entered a
10050   null response.  If he did, it immediately
10060   stops; otherwise, it attempts to treat the
10070   input string as a number in decimal, and
10080   returns to its caller with one of two things
10090   on the stack, either:
10100   <value of entered number> ... as "nos"
10110              0              ... as "tos"
10120   or:
10130   <original response string>...from "nos" down
10140              1              ... as "tos"
10150   (the latter applies in the event that the
10160   response string was not a valid decimal
10170   number.)
10180 rem
10190 inpdec: print input # 0 = if stop end
10200 0 ; 1 for fn 2 + ↑
10210 48 - # 0 < ; 9 > or if clr . . 1 return end
10220 % 10 * + next
10230 % 1 for % . next 0 return
```

Example 3: A Program Using Doubly-Nested For/Next Loops

```
100 ******************************************************
110 *                                                    *
120 * this program prints out the multiplication tables *
130 *           for products up to 9 * 9                 *
140 *                                                    *
150 ******************************************************
160 rem
200 "***" print 9,1 for fn cvtdec & print next 13,1 print
210 "***" print 36,1 for "." print next 13,1 print
220 9,1 for fn cvtdec & . . . 2 print ":" print
230 fn 9,1 for # fn * cvtdec & print next . 13,1 print
240 next stop
300 rem
310 this routine converts the number on the stack into
320   a decimal string and then pads it on the left with
330   blanks so as to make a string exactly four charac-
340   ters long.  Note: this routine will not work if the
350   input number is greater than 999.
360 rem
370 cvtdec: str$ 4 % - 1 for 32 next 4 return
```

Example 4: A Recursive Routine

```
4500 rem
4510 this routine replaces the quantity on "tos" with
4520   its factorial.  Note that the largest factorial
4530   less than 32768 is 7!, so this routine will not
4540   work if the input on "tos" is greater than 7.
4550 rem
4560 factorial: # 1 - # if factorial & * then . end return
```

Example 5: Combining BASIC Source with RPL Source

```
100 rem the rpl portion of this program may be compiled
110 rem and executed just as if this basic portion were
120 rem not here...
130 poke 32768+rnd(1)*2000,rnd(1)*256 : goto 130
140 *
150 * ...and, at the same time, the basic portion of this
160 * program may be run just as if this rpl portion
170 * were not here!  the "rem" in line 220 causes the
180 * rpl compiler to treat the entire basic portion as
190 * a comment, while basic never really has to deal with
200 * the rpl portion due to the "goto 130" in line 130.
210 *
220 rem
230 test: rnd rnd 2000 \ 32768 + poke test goto
240 rem
250 * one word of caution: don't try this with a basic
260 * program that has a line in it with a "rem" standing
270 * by itself.  the rpl compiler would then start
280 * trying to compile your basic program, which could
290 * cause it to become very confused.
300 rem
```

Example 6: An RPL Line-Renumbering Program in RPL

```
1000 you may have noticed that the rpl compiler pays
1010   absolutely no attention to line numbers except for
1020   the purpose of printing out error messages.  Thus,
1030   a line renumbering program for rpl source is
1040   utterly simple: all it needs to do is to chase
1050   through the line pointer chain, changing the line
1060   numbers as it goes.  this program renumbers
1070   the lines of any rpl program in memory (itself or
1080   any other) to start at 1000 and go up by 10's.
1090 rem
1100 1025 start @ loop: ; @ if % ; ; 2 + !
1110 @ % inc @ + loop goto end stop
1120 start: [1000] inc: [10]
```

A Final Bit of Advice for the Newcomer to Computers

Even though we have endeavored to be as clear and explicit in what is written above as we can, there is still, no doubt, a fair amount that went sailing right by you the first time around. Don't let this bother you: try out a few of the examples until you get the hang of it; then try modifying the examples a little. When you start writing your own programs, start them off simple, and upgrade them as you go along, testing what you have at various points along the way. The RPL Compiler is no playtoy: in order to get the speed and power of a language like this, certain sacrifices must be made in terms of ease of use and freedom from worry.

In the BASIC programming you have done up until now, BASIC has always slapped your hand if you got it too near the fire... if you went out of bounds on some array, or if you said "next" without having first said "for", it told you so, in no uncertain terms, and it pointed out the line you should look at to find the problem. But the price you paid for BASIC to hold your hand like that was that many of the programs you wrote would execute at only one-one-hundredth of the speed they might have executed at, had they made full use of the power of the microprocessor in your machine.

RPL does not by any means claim to offer you a hundred-fold speed increase. It occupies a useful mid-range in the tradeoff between ease of use and speed-efficiency, and offers you considerable memory-space savings in the bargain. But there are many errors you can make that RPL will not catch; many, even, that will drive you to utter distraction as you try to ferret them out. If you say "next" without having said "for", for example, the compiler will not bat an eyelash: your program will simply go quietly insane at the point where this happens, and the only indication you will get that something went awry is the obscure bit of gibberish "p!" -- and sometimes not even that.

But have faith when this happens, and don't go running to the nearest bridge to throw the whole RPL package (or yourself) into the drink... nobody ever said computers were trivial things to master. Simply reset your machine by turning it off and back on again, reload the compiler module and your source again, and trace through your program by hand to see if you can tell where it went wrong. If you decide after doing this that your program "has to work", that there is no way it could have done what it did, then try explaining to someone else exactly why that is the case -- 80 percent of the time, in the middle of your explanation your eyes

will suddenly light up, you will go "aaah...haaaa...!", and, with an entranced look, you will return to your computer to vanquish the bug for good. The other 20 percent of the time, try inserting some statements in your program to print out any suspicious quantities at various points along the way. If the problem is still no clearer after that, try isolating what appear to be the offending routines: write a separate little test program to pass known data to them and see what happens. Glance through Appendix F from time to time... you might even try reading through the rest of the manual again to see if there's something you missed. If after trying various combinations of these approaches for days and days on end, the program still won't work, may we suggest that you try your hand at riflery, or automobile maintenance. You know what they always say: "To err is human; to <u>really</u> foul things up requires a computer."

A Final Bit of Advice for the Newcomer to Computers

Even though we have endeavored to be as clear and explicit in what is written above as we can, there is still, no doubt, a fair amount that went sailing right by you the first time around. Don't let this bother you: try out a few of the examples until you get the hang of it; then try modifying the examples a little. When you start writing your own programs, start them off simple, and upgrade them as you go along, testing what you have at various points along the way. The RPL Compiler is no playtoy: in order to get the speed and power of a language like this, certain sacrifices must be made in terms of ease of use and freedom from worry.

In the BASIC programming you have done up until now, BASIC has always slapped your hand if you got it too near the fire... if you went out of bounds on some array, or if you said "next" without having first said "for", it told you so, in no uncertain terms, and it pointed out the line you should look at to find the problem. But the price you paid for BASIC to hold your hand like that was that many of the programs you wrote would execute at only one-one-hundredth of the speed they might have executed at, had they made full use of the power of the microprocessor in your machine.

RPL does not by any means claim to offer you a hundred-fold speed increase. It occupies a useful mid-range in the tradeoff between ease of use and speed-efficiency, and offers you considerable memory-space savings in the bargain. But there are many errors you can make that RPL will not catch; many, even, that will drive you to utter distraction as you try to ferret them out. If you say "next" without having said "for", for example, the compiler will not bat an eyelash: your program will simply go quietly insane at the point where this happens, and the only indication you will get that something went awry is the obscure bit of gibberish "p!" -- and sometimes not even that.

But have faith when this happens, and don't go running to the nearest bridge to throw the whole RPL package (or yourself) into the drink... nobody ever said computers were trivial things to master. Simply reset your machine by turning it off and back on again, reload the compiler module and your source again, and trace through your program by hand to see if you can tell where it went wrong. If you decide after doing this that your program "has to work", that there is no way it could have done what it did, then try explaining to someone else exactly why that is the case -- 80 percent of the time, in the middle of your explanation your eyes

enclosing the string in double-quotes and placing that between either square brackets or parentheses. This can be very handy for, among other things, forming predefined symbol tables for use in decoding user responses via the "fndsym" and "dcdrsp" routines globally defined in the compiler (see Appendix G). These strings are formed with the leading byte being the string length, and the string body following that.

Single-Quotes

If you write any programs that look at source entered via the screen editor, you may need to form tables and whatnot containing literal transcriptions of what the screen editor deposits in memory when lines containing BASIC keywords are entered. The compiler will take any material enclosed in single-quotes and copy it directly, byte for byte, from the source to the object. In case this strikes you as a frivolous feature and a waste of memory, consider that it costs exactly four extra bytes in the size of the compiler, and it makes programs like assemblers and compilers for Commodore machines much more readable (which will be a consideration if you ever decide to purchase the compiler source).

Spaces, Commas, and BASIC Keywords

You have presumably noted by now that both commas and spaces may be used just about anywhere to separate RPL keywords from their arguments, etc. What is not clear from the examples is that they are not by any means required (in fact, it is the fact that they are by and large not required that makes the compiler do 2 passes instead of just 1). The reason all the examples in the first section of the manual use them is that, if you don't, there are a few little "gotcha's" you have to look out for, arising out of certain things in the screen editor. If you have been using BASIC for a while, you will be able to figure out what they are: "print" or "input" immediately followed by "#" gets taken as a single entity, and will get flagged by the compiler as an unresolved reference; a reference to a symbol called "str" immediately followed by "$" has the same problem, except that the compiler won't even burp; in fact, contiguous references to any two symbols which when concatenated form a valid BASIC or RPL keyword will give strange results. If you wish to completely eliminate all possibility of such problems, feel free to use as many spaces and commas as you like; those who are more adventurous may cut down on the size of their source considerably by compressing most of these separators out, and almost never experience difficulty as a result. Be aware if you do this, however, that there are certain, obvious places where separators are required: between the

beginning of a symbol definition (not reference) and any alphabetic material that may precede it on the same line (see "unres ref", Appendix C); preceding a colon that has the function of restoring the compiler's location counter to its previous value (see below); and between an invocation of the subtraction operator and the pushing of a literal numeric, if in fact the simple pushing of a negative number is not desired.

Compile-Time Constants

Most assemblers have a facility, generally a pseudo-op called "EQU", which allows the programmer to assign an arbitrary value to a symbol, whereas most high-level languages, even compiled ones, don't. Well, being a sort of hybrid between assembly and high-level languages, RPL does, but it has limitations. By saying ":symbol:<value>:", where <value> is always a literal numeric constant, you can assign the value to the symbol in such a way that, anyplace you otherwise would have had to simply state the literal numeric constant, you may substitute the symbol name instead. For example, to get the first byte in the tape-2 buffer onto the stack, you could say "826,peek"; or, you could say "taptwobuff,peek", and, somewhere else in your code, say ":taptwobuff:826:".

The Back-ORG, and Undoing It

The above compile-time constant defining facility is a special case of a more general, and useful, feature. You recall that the RPL Compiler compiles from back to front, bottom to top, whatever. One reason it does this is that it is easier that way to build code downward from some fixed upper boundary, as opposed to building upward from some fixed lower boundary which is what BASIC tends to do. When you specify to the compiler that you want certain code or data placed into a particular spot in memory, then, what you specify is the upper boundary you want that code or data to be flush against. You do this using the colon character, but you precede the colon with a literal numeric constant instead of an alphabetic symbol (hence the restriction to all-alphabetic characters in symbol names). When the compiler sees that what immediately precedes the colon is numeric rather than alphabetic, it says to itself, "Ah, this guy wants me to stop placing the object code I generate where I have been placing it, and to place all further code from this point on, until told otherwise, extending downward from the point in memory whose address precedes the colon." Note that nothing will get poked into the byte at the specified address: the first poke, if there is one, will go into the next byte below it. In assembly language, this type of facility is generally available through a pseudo-op called "ORG",

and it affects code placement following the pseudo-op... this facility affects code preceding the pseudo-op, hence the term "back-ORG".

Now, ordinarily you are not going to want to have to worry about where in memory your object code resides. In fact, the back-ORG facility is primarily useful for defining compile-time constants, or perhaps for setting up machine-language routines in the tape-2 buffer, that sort of thing. Hence it would be desirable for you to have some way to tell the compiler, "Hey, I'm done doing this thing that I did the back-ORG for: let's go back to wherever we were before and start generating code there again." The way you do this is by simply putting in a colon where you want this to happen, making sure that the colon is immediately preceded by neither an alphabetic nor a numeric character (nor another colon, as will become clear below). Now you can see how the compile-time constant definitions work: reading backwards, you are saying "back-ORG to such-and-such a place, define a symbol there, and then restore the original location counter value." This location-counter "stack", by the way, is only one deep: don't try to back-ORG from within a back-ORG'ed region and expect two standalone colons to get you back the original location counter. Not that you would ever want to.

Global Symbols

To make a symbol global, all you do is define it using a double-colon instead of a single one. What this does for you is that once you have compiled a program, or piece of a program, containing symbols defined as global, those symbols may be referred to from another program entirely that you compile later. The only things you have to ensure in order for this to happen are that (a) the object code of any routine you define globally does not get clobbered before it is used (which you would ordinarily accomplish by specifying the "continue" option in response to the "enter object destination:" prompt when compiling subsequent modules), and (b) that you preserve the contents of the global symbol table from one compilation to the next (which you would ordinarily accomplish by specifying the "continue" option in response to the "enter gbl sym tbl option:" prompt when compiling subsequent modules).

A couple of minor points to remember when dealing with global symbols: if a local symbol and a global one happen to have the same name, the local one will override the global, i.e. the address that the local symbol is defined at will be the one that all of the references end up pointing to. Also keep in mind that the globalness of global symbols is a one-way affair: an unresolved reference in one compilation cannot be resolved by

eventually defining the symbol in a subsequent compilation.

The "Enter Object Destination:" Prompt

In the discussion of the back-ORG above, it was mentioned that you will ordinarily not want to have to get involved in determining where in memory your object code is to go. On the other hand, you do need a certain amount of flexibility in this respect, for example: how do you ensure that one of your object modules does not overwrite another when making use of the global symbol table to link modules together? How do you create a final, saleable product module (if that's what you're into) which does not contain the compiler itself and all the other rubbish that your user is not going to need? How can you compile a program that is too big to allow you to keep both the source and the object in memory at one time?

The answers to all these questions, and more, is found through examining your possible responses to the "enter object destination:" prompt. These are:

1. DEFAULT
   This option always starts building your object just below the lowestmost point of the compiler or whatever other associated Samurai Software products you have installed in your machine. Basically, the compiler and these associated products start at the top of memory and grow downward, and by specifying the default destination for your object you simply continue the progression. This is of course the option you should use unless you are doing something fancy, which is why it is the default.

2. CONTINUE
   This option starts building your object just below the start-execution address of the previous module you compiled (the reason we say the start-execution address and not the lowestmost point is that your previously compiled module may have done back-ORGes someplace). This is the destination you should use for all but the first module in a series of modules that you wish to link together using global symbols.

3. REPEAT
   This option starts building your object exactly where it started building it during the previous compilation. This option only differs from the default if you specified something other than "default" in the previous compilation: in particular it is useful if, having compiled your first module in a globally-linked series, you compile, say, the second or third, only to find that they contain errors.

Using the "repeat" option, you can correct the errors in the source and recompile without losing the object modules you had done so far in the series. The use of this option can be a powerful aid in program development: you can write a few routines, debug them, and make their entry points global; then delete their source from memory and begin working on the next level of routines, at the first compilation saying "continue", then saying "repeat", "repeat", "repeat", until you get the new ones right, at which point... and so on. After all, why keep compiling and recompiling fully debugged routines?

4. SOURCE
   This option starts building your object right on top of your source (the object will always take up much less space than the source except in pathological cases). Use of this option is one way to get around the problem of programs too big to permit both source and object to fit in memory at the same time (the other way is of course to break up the program into smaller pieces and link them together using global symbols). This option is a little dangerous in that, when compilation is complete, if by mistake you tell BASIC to list your program you could very well end up crashing your machine. It is recommended that you immediately issue the "new" command to BASIC upon completion of any compilation using this option.

5. COMP
   This option builds your object right on top of the compiler. It should only be used when memory space constraints become a real problem. It wipes out the compiler, but leaves the global symbol table (and the Samurai RPL Symbolic Debugger, if you have it) intact, so that you have the possibility of doing debugging and/or continuing with a globally-linked series. How do you continue with a globally-linked series if the compiler has been overwritten, you ask? Good question! The one thing you <u>don't</u> do is load in the next module and say "compile". Instead, you use the address limits shown upon completion of the first compilation to create a load module on tape or disk using the TIM or a "sys" to 63153 (see "Saving Load Modules" below). Then you reload the compiler, except you don't reload from the usual file, you load from the special file "justcplr" on the tape or diskette supplied. This special file contains "just the compiler", i.e., it does not overwrite the global symbol table, the way the usual compiler module does. Do a "new" after loading "justcplr", then load in the next piece of globally-linked program and say "compile", replying "continue" to both prompts. When it finishes, repeat the whole process for the next module, and so on. When the last

module is done, load all your little object module pieces back into memory, one after the other. Your complete object module is now ready to go: say "go" or "simulate". If this all sounds inordinately complicated, we ask you how else you propose to create an almost-7K object module on an 8K machine (or a 31K module on a 32K machine, etc.).

6. FINAL
This option builds your object right up flush against the bottom of the RPL "run-time library", and represents truly the limit in terms of memory space efficiency. This is the option you would use to create a saleable object product. The object module created through the use of this option has nothing more in it than what is needed to execute it: no symbolic debugger, no global symbol table, and no little beast in the background for handling the "compile" and "go" commands and keeping the screen editor from collapsing program lines down to just one leading space, etc. If your product does not contain any global symbol definitions or references, you can simply compile it, specifying "final" as your object destination (the compiler will not ask you about what to do with the global symbol table). If your product <u>does</u> make use of the global symbol table, things get a little more complicated. When you give "final" as an object destination, the reason that the compiler does not ask you where it should put the global symbol table is that it knows that the only place it <u>can</u> put it is up on the screen. Therefore you have to make sure, as you go through the series, that the various commands you issue and the machine's responses to them do not cause the screen to scroll or the table to be overwritten, otherwise your global symbols will get all messed up. "Going through the series" of compilations is carried out essentially similarly to the procedure under response 5, "comp", with the exception that, after the first module piece is compiled, the little beast that processes the "compile" command will be gone, so you will have to "sys" to location 0 in order to fire up the subsequent compilations. Note: this is the <u>only</u> circumstance under which the compiler will affect the contents of locations 0, 1, and 2, so your existing "usr" functions for the most part do not have to worry about getting clobbered. See below under "Saving Load Modules" for information on how to fire up the resultant complete load module.

7. A Decimal Number
If, for some reason, none of the above will place your object where you want it, you have the ultimate option of simply specifying an address at which it should begin being built. If the reply you give would place your object above

the "default destination" point, it will automatically be initially stored lower in memory and then moved up after compilation is complete, the same way it is when the "comp" and "final" destinations are specified. Please note that you are only accorded this protection for code in your program within the "default space", i.e. back-ORGed material is poked as it is encountered during pass 2.

8. The Null Response
   If you change your mind about wanting to run the compiler, you can simply wipe out the "default" reply by spacing over it, and hit RETURN: this will halt the compiler and return you to BASIC. You can also of course halt the compiler at any time by hitting the RUN/STOP key.

The "Enter Gbl Sym Tbl Option:" Prompt

Much of the time, particularly when you are just getting acquainted with RPL, you will not be using the global symbol table, and you will not want to have concern yourself with what might or might not be in it. On the other hand, when you are using global symbols, you need to have some way to tell the compiler when to clear the global symbol table out, when to leave it alone, and when to ignore the symbols introduced into it during the preceding compilation.

These functions are all available via the various responses you can give to the "enter gbl sym tbl option:" prompt, which are:

1. CONTINUE
   This option tells the compiler to leave the global symbol table alone as it prepares for compilation, and to simply continue adding global symbols to it during the present compilation, if there are any. The reason this is the default option is that ordinarily there won't be any, and by saying "continue" here, you continue to preserve the global symbols created during the compiler's compilation of itself (see Appendix G). This is also the option you would most likely use throughout any series of globally-linked compilations.

2. CLEAR
   This option tells the compiler to clear out the global symbol table in preparation for the present compilation. This is the option you would use for the first compilation of a globally-linked series, assuming none of the modules in the series uses routines in the compiler, in order to allot yourself the maximum amount of room in the table for your own global symbols.

3. REPEAT
   This option tells the compiler to, essentially, backtrack in the global symbol table to the point the table was at at the beginning of the preceding compilation, before proceeding with the current compilation. This is the option you would use in conjunction with the "repeat" response to the "enter object destination:" prompt, if the module you had just compiled contained global symbol definitions (if you "repeat" the compilation without "repeat"ing in the global symbol table, you will get the "duplicate symbol" error message on each of the global symbols defined in the current source).

4. A Decimal Number
   In case none of the above options will do, you can always simply specify where in memory you want the top of the global symbol table to be considered to be, by giving a literal address in response to the prompt. There is only one case we are aware of in which this is useful: the case in which you need to put the global symbol table up on the screen because your symbols overflow the space preallocated for global symbol storage (see "symbol table overflow", Appendix C). This case could have been covered by setting up another option in this list called "screen" -- but come on, everybody knows where the screen is: why waste memory?

5. The Null Response
   If you change your mind about wanting to do the compilation when you get this prompt, simply wipe out the "continue" by spacing over it, and hit RETURN. Unlike the corresponding action on the "enter object destination:" prompt, doing this will leave certain pointers and whatnot in peculiar states: keep in mind that if on the next compilation you specify the "repeat" object destination, you will be repeating the destination you gave in the present, aborted compilation (if you don't use "repeat" the next time around, you should be fine).

Saving Load Modules

Whether or not you do so for profit, you will probably, at some point, want to make disk or tape copies of object code, so that once you have debugged your programs you do not have to recompile them every time you want to run them. This can be done in a variety of ways.

The simplest is to get your program completely compiled, and then make a load module extending from the lowestmost point of the object you have created up to the upper limit of the memory in

your machine. If your machine has in it a ROM version containing the TIM, this is very easy -- on the CBM 8032 or 4032, the sequence might go something like this:

```
compile
rpl compiler (c) 1981 by tim stryker
enter object destination: final
0 errors: code goes from 29380 ($72c4) to 31531 ($7b2b)
ready.
poke 634,0:sys 634              (there must be a better way!)

b* pc bullshit
.s "module name",<device #>,72c4,8000
.x
ready.
```

Then, all you would need to do to load and run it is:

```
dload "module name"             (or 'load "etc."' for cassette)
searching for 0:module name
loading
ready.
sys 32734
```

(Replace the sys to 32734 with a sys to 16350 on a CBM 4016 or other 16K machine, or a sys to 8158 on an 8K machine -- these addresses, incidentally, will not change from one release of RPL to the next.)

This method has the advantage that you do not need to have previously loaded and run the compiler module in order to run the program. If you want to be able to run it by saying "run" instead of sys to someplace, do a "new" after you have compiled it, then write a line 10 containing the appropriate sys and save the whole shooting works from 0400 up to the top of memory. Or, write a little for-next loop to copy it down from high memory to just above the BASIC code, make the BASIC code include a loop to copy it back the other way before doing the sys, and save only from 0400 up to the top of the copy (be careful you don't overwrite the for-next variable you're using to do the copy).

When making tape or disk copies of object module pieces during a globally-linked series of compilations that started with an object destination of "comp" or "final", save only the portion of memory given at the completion of each compilation. The upper bound indicated in the compilation completion message is actually one greater than the address of the highest byte stored, so you can (in fact, must) use this address as the upper-bound argument in the save command to the TIM.

If your ROM version does not contain the TIM, then, as they say in the trade, you got a problem. As far as we know, only the version 1 ROMs lack the TIM. With the version 1 ROMs, the following type of procedure can be used (this example is on an 8K PET):

```
COMPILE
RPL COMPILER (C) 1981 BY TIM STRYKER
ENTER OBJECT DESTINATION: FINAL
0 ERRORS: CODE GOES FROM 5578 ($15CA) TO 6951 ($1B27)
READY.
POKE 229,39              (low-order byte of $1B27)
READY.
POKE 230,27              (high-order byte of $1B27)
READY.
POKE 247,202             (low-order byte of $15CA)
READY.
POKE 248,19              (high-order byte of $15CA)
READY.
POKE 238,0               (a filename length of zero)
READY.
POKE 241,1               (ensures use of tape #1)
READY.
SYS 63153                (invokes the tape-save routine)
```

Interfacing RPL to BASIC

Sooner or later every RPL programmer wants to be able to write a BASIC program that treats an RPL program as a subroutine. The RPL "stop" operator acts as an unconditional stop, returning the machine to immediate-mode BASIC, not to the calling BASIC program -- how, then, can this be accomplished?

You might suppose that you could try making your RPL subprogram sys to a machine-language routine that did a couple of PLA's and an RTS when it wanted to return to its BASIC caller. You cannot: one reason for this is that RPL ordinarily completely clears the hardware stack when it fires itself up. This is done in order to give your program the maximum stack space possible, and also in order not to confuse novices who are using the RPL Debugger (they would otherwise be forced to contend with the visual display of a lot of crap on the stack that they didn't put there).

However, there is a way to disable this initial stack clearing operation. The sys points mentioned above (32734 for a 32K machine, 16350 for a 16K machine, and 8158 for an 8K machine) contain code that loads Y with a 14 and then jumps to the main RPL fire-up sequence. The first three bytes in the main RPL fire-up sequence do a LDX with #255 and a TXS: this is what clears the stack. So, all you have to do to disable this is to poke a new

jump address in at the point 3 bytes beyond your original sys point, where this new jump address is 3 greater than the one that was there before.

Having done this, your RPL program still cannot just RTS its way back to BASIC, because RPL uses certain zero-page locations in strange ways, and it has to put them back the way they were. To facilitate this, RPL opcode 233 has been provided: to return to your BASIC caller, get the parameter stack back to its initial state, and then cause execution to pass through a byte containing a decimal 233. This is most easily accomplished using parentheses: just put a "(233)" in the execution path where you want this to happen. Alternatively, you could get a little fancier by defining a symbol, say, "rts" with a value equal to 233, and then say "(rts)". Either way, RPL execution will terminate, and BASIC execution will resume with the instruction following the sys.

Associated Products

Samurai Software makes a number of associated software products, designed to help you make the most of your investment in Commodore hardware. Among them are:

The Samurai RPL Symbolic Debugger Package

This package, invoked via the "simulate" command from BASIC, allows you to debug RPL programs using an interactive, "soft-key-driven" execution simulator. The keys used as "soft-keys" are those in the numeric keypad, and a small matrix of legends is kept constantly present in the lower right corner of the screen, making the package extremely easy to use. Features included are:

a) Constant display on the right-hand side of the screen of the complete contents (up to 18 entries deep) of both the parameter stack and the return stack as execution progresses.

b) The ability to single-step through your RPL program at any point, watching as data is added to and removed from the stacks. The operator or operation to be executed on the next step is displayed as it appears in the source (except for pushes and if-then-end operations) at all times.

c) The ability to set breakpoints in your RPL object and to reach them via either a "fast-single-step" mode or a straight "execute-until-hit" mode.

d) The option to suspend execution on any operation boundary and to then proceed to edit both stacks, with full visual feedback on the screen.

e) The ability to examine memory in hex and "dis-compiled RPL" at any time.

f) The ability to specify addresses for use in break-pointing, memory display, etc. in terms of arbitrarily complicated expressions which may contain references to symbols from both the global symbol table and the most recently created local symbol table.

g) A host of extra run-time error checking functions, such as checking for "next" without "for", "return" without "&", return stack overflow, execution of illegal RPL "opcodes", and the like.

h) Virtually complete transparency of execution simulation: i.e. the debugger uses no additional zero-page memory, no tape buffer memory, and, of particular importance, it does not interfere at all with the simulated program's use of the video display (this is implemented via a screen-memory swapping scheme).

The Samurai 6502 Assembler

This program takes 6502 assembly source input via the ordinary screen editor, assembles it, and outputs either object to memory or an assembly listing to printer or both. No printer is required to make use of the package, although having one certainly makes it nicer. It supports the standard MOS Technology opcode and operand syntax, and includes the following features:

a) Full "precedence parsing" of expressions, including modulo, exponentiation, and logical comparison operators, up to 10 levels of parentheses, and quantities expressed either symbolically or in hex, decimal, binary, or ASCII form.

b) Symbol names up to 50 characters long, stored in variable-length format for maximum space-efficiency (symbol names may contain numeric characters).

c) The same local and global symbol table concepts used in the RPL compiler, enabling the sharing of symbols between the two languages with a minimum of effort, and facilitating the generation of true subroutine libraries in assembly language.

d) Both ORG and "back-ORG" pseudo-ops, plus a default memory allocation concept similar to that in the RPL compiler, permitting easy integration of assembled routines with both BASIC and RPL drivers.

e) Macro and conditional-assembly handling capabilities, including the ability to place macros into the global symbol table for use in more than one assembly.

f) A full array of pseudo-ops for such purposes as controlling printer output, creating "dummy sections", generating symbol value messages at assembly-time, and the like.

The RPL Compiler Source Package

This product consists of a diskette containing the documented source to the RPL Compiler and whatever other associated Samurai Software products you may have (the price of the package depends on what other products you have). You must have a CBM 8032 in order to make use of this package.

Since the compiler is written in RPL, it can, and does, compile itself. This product is offered to assist those individuals who have the irrepressible urge to tinker: features may be added to or deleted from any of the Samurai utilities using this package. It may also serve as an aid to instruction for those interested in learning more about compiler design for small computers. Note that this package contains only the compiler and associated products sources themselves, not a great deal of explanatory material or the source to the RPL "run-time library", the set of assembly routines actually implementing the RPL operators.

Appendix A: RPL/FORTH Differences

RPL and FORTH are similar in their use of RPN, their use of both a parameter stack and a return stack, and their optimization for structure, speed, and compactness. However, they differ in a number of other respects. These differences arise primarily out of differences in design approaches:

1. FORTH object code is "threaded", whereas RPL object uses a form of "p-code". This has resulted in RPL being substantially better at conserving memory space than FORTH, even though the two languages run at comparable speeds. Most RPL operators take only a single byte of object space to invoke, and most numeric push operations take up only one or two bytes, whereas FORTH operators take up a minimum of two bytes apiece, and a numeric push in FORTH frequently takes up four full bytes. At the same time, preliminary benchmarks indicate that, if anything, a given RPL program will run slightly faster than its FORTH equivalent.

2. The RPL Compiler was designed specifically for the Commodore series of computers, whereas FORTH was designed to be fairly machine-independent. Thus, although FORTH is more highly transportable, it is also inherently considerably clumsier on Commodore machines. In particular, when using RPL you also have the full power of Commodore BASIC at your disposal, including the screen editor and all file I/O, whereas FORTH takes full control of your machine, with its own peculiar editor and its own special file I/O functions (which are not nearly as complete as BASIC's). Not only does this mean that in order to use FORTH you have to learn a whole new set of operating system command syntaxes, but it also means that a fair amount of memory space is consumed in the FORTH monitor duplicating capabilities which are already present in the machine -- space which could certainly be put to better use (like, for storing your programs, etc.).

3. FORTH is set up as a kind of hybrid interpreter/compiler, which has definite implications in terms of the order in which it must receive its source code statements. FORTH forces you to enter code "bottom-up", and source listings generated in the standard FORTH manner will always come out with the lowest-level routines at the start of the listing, proceeding down until the program's mainline is found at the very end. RPL, being more of the usual type of straight compiler, permits you to arrange your code more naturally, with the mainline at the start, and with successively lower-level routines found progressively later in the listing. Even so, RPL does not force you into placing your

lower-level routines in any particular order -- as with BASIC, FORTRAN, and the like, as long as the routines are present in the compilation, they will be found.

4. FORTH's "threaded" object structure makes it inherently very difficult to come up with a debugging package along the lines of the one available for RPL... which is why no such package is available for FORTH. RPL's "p-code-driven" object structure lends itself nicely to the possibility of doing extensive automatic run-time error checking in a debug mode while at the same time keeping run-time overhead at an absolute minimum during normal running.

5. RPL recognizes that the one complaint users almost invariably have about their systems is that they don't have enough memory (this applies as much to the CBM 8000 series user with 96K as it does to the PET-2001 user with 8K). Thus it endeavors to give you the utmost functionality possible in the memory you have. Unlike FORTH, it does not require keywords, symbol references, and whatnot, to be separated by spaces in the source; also unlike FORTH, it allows both its own keywords and user symbol names to be compressed by the BASIC screen editor, conserving both source and symbol-table space. In addition, it supports numerous schemes for dealing with memory space constraints, including the local/global symbol table arrangement, options to allow the object to overwrite either the source or the compiler itself, and the option to place the global symbol table on the screen if necessary.

6. FORTH implementations generally support some sort of assembler embedded within the FORTH package itself. This is a nice concept, but the assembler tends to be of marginal utility since it uses a bizarre Reverse-Polish syntax for the actual assembly instructions. RPL was designed to interface easily with assembly language, but not to try to incorporate an assembler: a normal, standard 6502 assembler using the identical global symbol table structure as RPL is available separately, if the user needs it.

7. Other differences between the languages are mostly minor ones concerning differences in syntax. Here is a random sampling of these:
   a) In FORTH you invoke a routine by simply stating the name of its entry point; in RPL you invoke a routine by pushing the address of its entry point onto the stack and then saying either "&" (call) or "sys".
   b) FORTH has various pseudo-ops for declaring symbols to be names of one- and two-byte variables, etc.; RPL treats memory allocation more the way an assembler does, insofar

as space allocated for variables is indistinguishable to the compiler from space allocated for program.

c) FORTH's "do...loop" construct is similar to RPL's "for...next" construct except that in FORTH the upper bound specified is 1 beyond the maximum loop counter value desired; RPL is more like BASIC in this respect.

d) FORTH implementations tend to support various flavors of looping constructs like "begin...until" and "while...perform...pend", whereas RPL does not (these structures may all be implemented using different combinations of RPL's "if...then...end" and "goto").

e) The FORTH operators for "top of stack duplicate", "swap top two stack entries", and so on, are generally intended to be spelled out in the source, as "dup", "swap", etc.; in RPL, because they are so frequently used, the source representations of the stack-management operators are single characters: "#" (dup), "%" (swap), etc. One of these differences deserves special mention: FORTH's "." operator means "print-numeric", whereas RPL's "." operator is the same as FORTH's "drop" (RPL's "print" operator is simply "print", available on CBM machines through the "?" character).

f) FORTH uses parentheses to delimit comments; RPL uses parentheses for array formation and other purposes, and uses pairs of "rem"s to delimit comments.

Appendix B: Binary, Hexadecimal, and 2's Complement

Digital computers store all numbers, letters, and entities of every kind in "binary", which is another word for "base two". In binary, there are only two possible digits: one and zero. To see how larger numbers are stored in binary, consider a number, say, 2058, in base ten. When you write "2058", you are really writing an abbreviation for "2 times ten-cubed, plus 0 times ten-squared, plus 5 times ten-to-the-first-power, plus 8".

Binary works the same way, except that the number the powers are taken to is two instead of ten. For example, when you write "1101" in binary, you are writing an abbreviation for "1 times two-cubed, plus 1 times two-squared, plus 0 times two-to-the-first-power, plus 1", which is 8+4+0+1, or 13 in base ten. The term "bit" is an acronym for "binary digit": it may be used in referring to any quantity or piece of a quantity that can take on only one of the two values zero and one. Many computers (the PET/CBM series among them) commonly deal internally with groups of eight bits at a time: the term "byte" refers to any one of these groups of eight bits.

Computer "memory" is generally organized as a long sequence of bytes. Each byte in the sequence has an "address", which is nothing more than a position-number indicating how far the given byte is from the beginning of the sequence. The byte with the lowest address is at address 0, the next one is at address 1, etc. You may think of addresses in any base you like, but at the lowest level, like everything else in the computer, they are encoded in binary. In the PET/CBM class of machines, addresses are sixteen bits long, which means that, in base ten, they may extend from 0 up to 65535. (In case you are interested, the PET/CBM stores special internal quantities in addresses 0 through 1023, and your BASIC and RPL programs from 1024 on up. At address 32768, the video display memory begins, and above that is the BASIC "interpreter" program itself, along with the screen editor program and various other things, which extend all the way up to address 65535.)

"Hexadecimal" is another word for "base sixteen", but it is more useful to think of hexadecimal as a method for abbreviating binary. In hex, each of the digits 0 through 9, and each of the letters A through F, corresponds to one possible pattern of four bits, as follows:

Binary	Hex	Binary	Hex	Binary	Hex	Binary	Hex
0000	= 0	0100	= 4	1000	= 8	1100	= C
0001	= 1	0101	= 5	1001	= 9	1101	= D
0010	= 2	0110	= 6	1010	= A	1110	= E
0011	= 3	0111	= 7	1011	= B	1111	= F

Hex numbers are generally written with a dollar sign in front, to distinguish them from "decimal", or base ten, numbers: thus, for example, "$40C2" is simply shorthand for the binary number 0100000011000010. To convert from hex into decimal, you can think of the hex "digits" A through F as representing the decimal quantities 10 through 15, and make use of the fact that hex is in fact base sixteen. For example, "$40C2" is "4 times sixteen-cubed, plus 0 times sixteen-squared, plus 12 times sixteen-to-the-first-power, plus 2", or 4*4096+0*256+12*16+2, which is 16578 in decimal. Note that the highest address it is possible to represent using sixteen bits is $FFFF, which is 15*4096+15*256+15*16+15, or 65535.

When a group of sixteen bits is used to represent an address, the above scheme is all well and good, because addresses are always thought of as being positive numbers. Computations, however, will frequently involve negative numbers. In "2's complement" notation, bit patterns $0000 through $7FFF are considered to represent positive numbers, in exactly the manner shown above, but bit patterns $8000 through $FFFF represent negative numbers. To form a negative number using 2's complement notation, you start with the representation of its absolute value in binary, you "invert" all of its bits (i.e., you change all the ones to zeroes and all the zeroes to ones), and then you add 1. For example, to form what in decimal would be -12, you start with the representation of 12 in binary (and let me use the hex abbreviation for that): $000C. Now you invert every bit in it to get $FFF3; then you add 1, and you have $FFF4, the 2's complement representation of decimal -12. This scheme effectively uses up the "high-order", or leftmost, bit of each number to encode the number's sign, plus or minus, so the range of possible positive numbers under it is now only 0 through 32767 decimal. The range of possible negative numbers becomes -32768 through -1 decimal, corresponding to bit patterns $8000 through $FFFF.

Appendix C: Compiler Error Messages

BAD TOKEN "<x>" IN LINE <###>

This message indicates that an illegal "token" has been found in the specified line. An illegal token may be something like an arithmetic operator found within square brackets, or it may be one of two BASIC keywords: "data" or "rem". Due to certain peculiarities in Commodore's screen editor, RPL cannot allow you to use the word "data" anywhere outside of comments, either by itself or as a substring of another symbol. For the same reason, it is illegal to use the word "rem" outside of comments except to delimit the beginning and end of comment sections. This need not be of concern to you, though -- the compiler will tell you if you violate either of these rules. To fix the problem, simply change your code so that the indicated illegal token(s) do not appear, and recompile.

DUP SYM "<xyz>" IN LINE <###>

This message (short for "duplicate symbol") indicates that you have attempted to define the specified symbol more than once. A given symbol name may be referred to as many times as desired in a program, but it may only appear once with a colon immediately following it.

IF WITHOUT END IN LINE <###>

Each time you invoke the "if" operator, there must be a corresponding "end" operator somewhere after it in your program. RPL does not assume that you only want the remainder of the line the "if" is on to be made conditional, the way BASIC does. This message indicates that, compiling from bottom to top, the compiler came across an "if" that it had not yet seen the corresponding "end" for. To fix it, put an "end" at the end of the portion of code you want made conditional upon the result of the indicated "if".

ILLEGAL QTY IN LINE <###>

This message indicates one of two things: either you have specified a literal number or symbol within parentheses which is negative or larger than 255; or, you have attempted a backward symbolic reference within parentheses. Remember that parentheses call for single-byte storage of the quantities you specify: numbers that are negative or greater than 255 will not fit in one byte, and backward symbolic references require two bytes for pointer storage during pass 2 of compilation. To fix

a problem of the first type, either truncate your number or allocate space for it using brackets; to fix a problem of the second type, move the symbol definition in question to some point past the last place where you refer to it inside parentheses.

MISSING) OR] IN LINE <###>

This message indicates that the compiler came across an open-bracket or open-parenthesis in the middle of what it thought was executable code. In all probability the way to fix it is to indeed, put in the missing) or].

ODD QUOTE COUNT IN LINE <###>

This message indicates that an odd number of double- or single-quotes was found in the specified line. Ordinarily, this is just a typo -- fix the line and recompile.

SYMBOL TABLE OVERFLOW!

This message indicates that your machine does not contain enough memory to hold your RPL source code, plus your RPL object code, plus the compiler itself and whatever other associated products you may have along with it, plus the local and global symbol tables. It does not tell you which table overflowed, but since the global symbol table cannot have overflowed unless you pretty well knew what you were doing anyhow, the expectation is that you will be able to figure out which one it was. If you cannot, then it was the local symbol table that overflowed. Possible solutions to local symbol table overflow include: (a) buy more memory; (b) cut down on the size of your source code by eliminating or shortening comments, symbol names, etc.; (c) break up your program into smaller modules and use the global symbol table to link them together; or (d) specify that you want the object to overwrite the source (reply "source" to the "enter object destination:" prompt). Possible solutions to global symbol table overflow include: (a) eliminate or shorten some of your global symbol names; (b) use the "clear" global symbol table option for the first compilation of your series; (c) put the global symbol table up on the screen (reply "32768" to the first "enter gbl sym tbl option:" prompt, and "continue" thereafter); or (d) buy the Samurai RPL Compiler Source package and recompile the compiler itself to define a larger global symbol table.

THEN WITHOUT END IN LINE <###>

The meaning of this message is similar to that of the "if without end" message above. Every "then" must be preceded by

its corresponding "if" and followed by its corresponding "end".

TOO FEW IF'S

This message, which can come only at the very end of compilation, indicates to you that you have more "end"s than "if"s in your program. The compiler has no way of knowing where you intended to put the missing "if"(s), or whether perhaps instead you typed in one "end" too many someplace. This type of error is always lots of fun to resolve.

UNRES REF "<xyz>" IN LINE <###>

This message (short for "unresolved reference") crops up whenever you attempt to refer to a symbol that is defined neither in your program nor in the global symbol table. It will generally be due to a typo in either the reference or the symbol definition itself. If it is not, make sure that you are defining the symbol properly, that is, that its definition does not appear within a comment or enclosed in quotes or brackets. Also, if the symbol definition does not appear at the beginning of the line it is in, make sure that it does not directly abut an alphabetic RPL keyword... e.g. "peeklabel:" will get taken as defining a symbol called "peeklabel", not as a "peek" followed by a definition of the symbol "label".

Appendix D: Space and Time Information On RPL Operators

The first column contains the operator or operation designation, the second the amount of object code memory space it uses, in bytes. The third and fourth columns give the best case and worst case execution times, respectively, in units of milliseconds (thousandths of a second). Testing was done on a machine with the Version 4 ROMs, and the execution times shown include interrupt overhead, so they may differ slightly for other ROM versions.

Oprtr	Bytes	Best(ms)	Worst(ms)	Remarks
+	1	0.083	0.083	
-	1	0.091	0.091	
*	1	0.279	1.135	Smaller operand should be at TOS, worst case when 31 > TOS > -31 is 0.554ms
/	1	1.133	1.204	
\	1	1.126	1.192	
str$	1	3.070	3.257	3.030ms + 0.040ms per digit (+ 0.035ms if negative)
chr$	1	0.251	0.255	
"..."	n+2	0.163	1.383	0.143ms + (20 * n)ms
print	1	0.066	2835.000	0.066ms + 0.214ms per character + 45.000ms per carriage return
>	1	0.093	0.097	
<	1	0.109	0.113	
=	1	0.082	0.093	
and	1	0.080	0.080	
or	1	0.080	0.080	
not	1	0.074	0.074	
#	1	0.078	0.078	
;	1	0.081	0.081	
↑	1	0.094	0.094	
%	1	0.173	0.173	
$	1	0.193	3.853	0.071ms + (61 * TOS)ms
.	1	0.061	0.061	
new	1	0.053	0.053	
int	1	0.076	0.076	
@	1	0.089	0.089	
!	1	0.092	0.092	
peek	1	0.082	0.082	
poke	1	0.083	0.083	
&	1	0.089	0.089	
return	1	0.082	0.082	
sys	1	0.084	0.084	Time includes the RTS back
goto	1	0.059	0.059	
if	3	0.149	0.149	
then	3	0.116	0.116	Executed at end of if-clause
end	0	-	-	Compiler pseudo-op

for	1	0.198	0.198	
next	1	0.101	0.106	
fn	1	0.071	0.071	
clr	1	0.062	0.062	
rnd	1	0.103	0.103	
get	1	0.095	0.265	
input	1	-	-	Not applicable
stop	1	-	-	Not measurable
push-6	1	0.057	0.057	6-bit push (-1 < x < 64)
push-15	2	0.057	0.057	15-bit push (63 < x < 32768)
push-16	3	0.091	0.091	16-bit push (x < 0, or bkwd reference)
(233)	1	-	-	Special exit -- not measurable
(234)	1	0.049	0.049	RPL no-op (unaccessible symbolically)

Appendix E: RPL Memory Usage

Machine-language programmers will be primarily concerned about memory utilization below address $0400. You may be relieved to know that RPL in no way disturbs the contents of either tape buffer at any time. In fact all of memory below $0400 is yours to do with as you please, with 4 exceptions:

1) The hardware stack page, decimal addresses 256 through 511, is used for both the parameter stack and the return stack. At no time should you try to use any portion of this page outside the stack context (not that it is likely you would want to).

2) The area in the zero page that BASIC uses for its "CHRGET" routine is swapped out to high memory when RPL fires itself up, and is swapped back when execution finishes. RPL needs this 30-byte block for stack pointers, program counter, and scratch memory for the various operators: you should never try to poke anything in here. This region extends from 194 through 223 under the Version 1 ROMs, and from 112 through 139 under all the rest.

3) The BASIC input buffer is used <u>at compile-time only</u> for storage of the if-then-end stack (this is the 80-byte area starting at address 10 under the Version 1 ROMs and at address 512 under all others). RPL also uses it during execution of your own program, but <u>only</u> during processing of the "input" operator.

4) Last, and least, locations 0, 1, and 2, get clobbered by the compiler <u>only</u> if you specify "final" as your object destination. This is a one-time event: a couple of milliseconds after you hit RETURN upon specifying "final" as an object destination, these three bytes become a JMP to the compiler's entry point, but that's it. At no other time does any Samurai utility touch these three bytes.

Above $0400, things are not quite so simple. Because the compiler and all of the other Samurai goodies are set up flush against the top of memory, their exact addresses will depend on the size of your machine. In addition, their sizes and precise layouts will depend both on the ROM version you have and on the particular collection of Samurai utilities you have purchased. In general, the following is about all that can be said with any assurance:

a) The 2 bytes at the very top of memory will contain the start address of the last thing you compiled.

b) The 2 bytes just below that will contain the start address of the debugger, if you have it in memory.

c) The 2 bytes just below that will contain the start address of the compiler, if you have it in memory.

d) The 2 bytes just below that will contain the start address of the assembler, if you have it in memory. This will also be the address which will become the upper bound on your object code when you compile something using the "default" object destination.

e) The global symbol table, when not up on the screen, will be located just above the debugger, and just below the RPL "run-time library": its address bounds can be found as discussed in Appendix G.

f) The local symbol table will always be located starting 14 bytes beyond the end of your source (this gives you room for a couple of miscellaneous BASIC variables between the end of your source and the l.s.t.). It grows upwards from there, while your object grows downward from high memory. The exact address bounds on the l.s.t. can be found as discussed in Appendix G. The l.s.t. and your BASIC variables are totally unprotected from one another, but this will not ordinarily present a problem.

g) MAXRAM (address 134 under the Version 1 ROMs; address 52 under all others) will always be automatically set either to the start address of the last program compiled, or to the assembler's start address, whichever is lower; thus, as long as your BASIC programs do not specifically poke anything above this limit, both they and your RPL programs will continue to work fine without any need for tinkering on your part.

h) The little sliver of memory between the end of the video display and the end of the 1K or 2K of RAM implementing it is used by the compiler for the little program that runs at the end of every compilation, transferring the object from where it has been built to where it is to be run (this is only needed for object destinations of "comp", "final", and the like). Therefore do not try to place object code here... the rest of the screen is fair game, of course, and can be amusing to build object code for.

Appendix F: Gotcha's and Common Usage Errors Unique to RPL

RPL represents a radically different approach to programming for many people, so it is to be expected that "careless" errors will crop up in the first few programs. This guide is intended to help the newcomer out by presenting a list of things to watch out for -- particularly those which RPL does not have in common with BASIC.

FORGETTING TO RE-COMPILE AFTER CHANGING THE SOURCE

Interacting with RPL is so much like BASIC in many ways that it may be difficult at first to remember to re-compile every time you make changes to your source code. This can lead to very perplexing situations ("But I'm sure I fixed that bug already -- what is it doing back here again?"). Just try to remember the cardinal rule of compiled languages: for purposes of testing, a change to the program is not really made until it is made to the object code -- changing the source is not enough.

SYMBOL NAMES INVOLVING DECIMAL DIGITS

Symbol names must be composed entirely of alphabetic characters. An attempt to define a symbol called, for example, "wazoo22", by sticking a colon after the name, will cause the compiler to think that you want to "back-ORG" to location 22, and to start building object code there. This can be disastrous: depending on the circumstances, it can even "crash" your machine, meaning that you will lose whatever you have in memory at the time. Just remember to compose your symbol names such that they are always entirely alphabetic, and you will be fine.

"REM" OR "DATA" APPEARING IN A PROGRAM LINE

Although the compiler will catch this type of error, it can be distressing because the use of either of these keywords can give rise to a whole slew of associated errors on the same line. If you happen to think of it, try not to choose symbol names like "userdata" and "remainder", that is, names that have either of these two strings embedded within them. If you do forget, don't let it bother you that every line on which one of these symbols appears gives you all kinds of strange error messages: just look for the message that says 'illegal token "rem"' or 'illegal token "data"'. Generally the only thing wrong with such lines is the use of the indicated token: change the symbol name everywhere it appears and compile again.

IMPROPER NESTING OF SUBROUTINES WITH FOR/NEXT LOOPS

A person accustomed to BASIC may take a while to get used to the idea that all "for/next" loops must be "terminated" in RPL. If this does not ring a bell, read back over the section on "for", "next", "fn", and "clr" in Part I. Another potential problem in this class is the sort of situation where one tries to say "fn" or "next" from within a subroutine, where the corresponding "for" was done before the subroutine was entered. Once you enter a subroutine, any "for/next context" you had available to you before entering the subroutine is strictly unavailable to you until you return from it. Saying "fn" illegally like this will simply cause your program to push a garbage value; saying "next" or "clr" will almost certainly "crash" the machine. Remember, both "for/next" information and subroutine return-addresses are stored on the return stack.

NUMBERS GREATER THAN 32767

Remember that RPL numbers are normally treated as being in the range from -32768 to 32767. Add 1 to 32767 and you'll get -32768. Multiply 10000 by 5 and you'll get -15536. Getting around this problem, if it is a problem, may take some effort.

"NOT" OF A NONZERO QUANTITY IS NOT ALWAYS ZERO

One sometimes uses the value of a counter of some kind directly as an argument to "if": if the counter is nonzero, the if-clause is executed, otherwise it is not. Then one is sometimes tempted to suppose that by "not"-ing the counter before doing the "if", the sense of the "if" will be inverted: this is not the case. "Not" inverts <u>all</u> the bits in the TOS; the only nonzero quantity it will convert to a zero is -1.

Appendix G: Global Symbols of Interest in Compiler

Since the RPL Compiler is written in RPL, it compiles itself. As a result of this compilation, a number of global symbols are left sitting in the global symbol table which may be of interest to you. You may of course refer to any of these symbols anywhere in your program, just as though you had defined them yourself. The text of each routine is shown here so that if you use an object destination of "comp" or "final", you can incorporate any routines you use into your own program. Where compile-time constants are shown, they apply to the Version 4 ROMs.

GETINP: A little prompt-and-get-input routine

```
3030 rem
3040 *  get kb input routine: expects prompt string on stack
3050 *     prints prompt, gets input, stops if null response
3060 rem
3070 getinp::print,input,#if,return,end,stop
```

CVTNUM: A decimal response-decoder routine

```
3210 rem
3220 *  convert number from ascii to binary routine:
3230 *     expects non-null string on stack
3240 *     pops arg, returns with either:
3250 *     <value of cvt'ed number>,0 ...if valid dec # found
3260 *                                1 ...otherwise
3270 rem
3280 cvtnum::0cerr,poke0%1for10*%48-#9>;0<or
3290 iflcerr,poke,end+next,cerr,peek#if%.end,return
3300 cerr:#
```

PUSHST: A push-string-from memory routine

```
2290 rem
2300 *  push string routine: expects ptr to string on stack
2310 *     pops arg, returns with string on stack
2320 rem
2330 pushst::#peek;+%peek0for#peek%1-next.return
```

FNDSYM: A symbol-table-lookup routine

Both FNDSYM and PUTSYM use a data structure called a "symbol table block". This is composed of 3 pointers, as follows:

offset 0: pointer to the current end of the symbol table

offset 2: pointer to the start of the symbol table

offset 4: pointer to the byte one beyond the end of the area allocated for the use of the table (this entry is only needed if symbols will be added to the table)

The symbol tables themselves are arranged as a sequence of strings and their corresponding values, the values being two-byte quantities. Note that this makes it very convenient to look up responses to prompts using FNDSYM, e.g.:

```
100 askyn: "need instructions? " getinp & yntbl fndsym &
110 if print " no good: yes or no? " print askyn goto end
120 # peek + 1 + @ 1 = if instruct & end
 ...
5000 yntbl: [yntend,yntbgn]
5010 yntbgn: ["yes",1 "no",0 "y",1 "n",0 "help",1] yntend:
```

```
4000 rem
4010 *  find symbol routine: (uses mach lang for hi speed)
4020 *     expects string, ptr to sym tbl block on stack
4030 *     searches spec'd tbl for stg, ret's with either:
4040 *     <ptr to sym tbl entry>,0 ...if found
4050 *        <original string> , 1 ...otherwise
4060 rem
4070 fndsym::#@temp!2+@
4080 csylp:#temp@<if0fsyhlp,sys,if%1for%.next0return,end
4090 csylp,goto,end.1return
4100 fsyhlp:(189,5,1,133,zpa,189,6,1,133,zpb,189,7,1)
4110 (141)[ctr](169,127,136,200,49,zpa,221,7,1,208,14)
4120 (232,232,169,255,204)[ctr](208,239,186,157,3,1,96)
4130 (186,160,0,177,zpa,41,127,24,105,3,101,zpa)
4140 (157,5,1,144,3,254,6,1,96)
4150 ctr:# temp:## :zpa:30: :zpb:31:
```

PUTSYM: A routine to make a new entry in a symbol table

```
3390 rem
3400 *  add sym to sym tbl routine:
3410 *     expects value, stg, sym tbl blk ptr on stack
3420 *     pops args, returns with 0 if ok, 1 if tbl full
3430 rem
3440 putsym::#temp!#@3↑127and+3+;4+@>
3450 if.127and0for.next1return,end
3460 @;127and0for;;pokel+%.next;;!2+temp@!.0return
```

BASPRT: A routine for outputting compressed BASIC text

```
3440 rem
3450 *  output basic text routine: expects string on stack
3460 *     pops and prints it, decoding kw's as it goes
3470 rem
3480 basprt::1for#128<iflprint,then,baskwt%128-#
3490 iflfor,chslp:1+#peek128<if,chslp,goto,end,next1+then.end
3500 prlp:#peek#127and1print128<ifl+prlp,goto,end.end,next
3510 return
3520 :baskwt:45234:
```

LOCST & GBLST: The addresses of the RPL symbol table blocks

The addresses of the local and global symbol table blocks are, themselves, available as global symbols. This may seem a little strange, but consider their utility in the following sample program for printing out the contents of both symbol tables:

```
100 ********************************
110 * symbol table display program *
120 ********************************
130 rem
140 "local " print locst outst &
150 "global " print gblst outst &
160 stop
1000 rem
1010 *  output symbol table routine:
1020 *     expects pointer to symbol table block on stack
1030 *     pops arg, outputs table, and returns
1040 rem
1050 outst: "symbol table" print 13 1 print # @ % 2 + @
1060 ostlp: ; ; > if # pushst & basprt & ": " print
1070 # peek + # 1 + @ # str$ print " ($" print chr$ print
1080 ")" print 13 1 print 3 + ostlp goto end . . return
```

Appendix H: RPL Quick Reference Sheet

To push a literal integer, simply state the integer.
To push a symbol value, state the symbol name.
To push a character string, enclose it in double-quotes.

RPL Operators:

+	add TOS to NOS	new	drop all
-	subtract TOS from NOS	int	interchg bytes in TOS
*	multiply NOS by TOS	@	fetch word at TOS addr
/	divide NOS by TOS	!	store NOS to TOS addr
\	NOS gets NOS mod TOS	peek	fetch byte at TOS addr
str$	cvt TOS to decimal stg	poke	poke NOS into TOS addr
chr$	cvt TOS to hex stg	&	call address on TOS
print	output string on TOS	return	return
>	true if NOS>TOS	sys	sys to address on TOS
<	true if NOS<TOS	goto	goto address on TOS
=	true if NOS=TOS	if	if TOS<>0 then
and	AND TOS into NOS	then	begin else-clause
or	OR TOS into NOS	end	end if-range
not	invert bits in TOS	for	for TOS to NOS
#	push another TOS	next	end for/next
;	push another NOS	fn	push for/next variable
↑	n-th	clr	clear current for/next
%	swap TOS with NOS	rnd	push random integer
$	rotate	get	push 1 byte from keybd
.	drop TOS	input	input stg from keybd
		stop	return to BASIC

Commas and spaces are not significant except as padding between successive numbers, operators, and pseudo-ops.
Enclose inline documentation between pairs of "rem"s, and put each "rem" on a line by itself.

Symbol names must be entirely alphabetic (no embedded numbers).
To define a local symbol, state its name immediately followed by a colon (no intervening spaces or commas).
To define a global symbol, state its name immediately followed by two colons (no intervening spaces or commas).
To "back-ORG", state a literal numeric address immediately followed by a colon (no intervening spaces or commas).
To restore previous location counter, insert a colon preceded by neither an alphabetic nor a numeric character.

To form an array of single-byte values, enclose a list of literal numbers and/or symbols in parentheses.
To form an array of double-byte values, enclose a list of literal numbers and/or symbols in square brackets.
To form a character string directly in memory, enclose it in double-quotes and enclose that in brackets or parentheses.
To allocate memory without specifying contents, enclose the byte count desired between angle-brackets within parentheses.