BASIC, FORTH, and RPL

by Timothy Stryker

BASIC and FORTH, two widely accepted high-level languages for microcomputers, are compared to RPL, a relative newcomer to the field. The languages are compared with respect to time-efficiency, space-efficiency, transportability, and ease-of-use considerations.

Although RPL is currently available only for PET and CBM, this article is of general interest.

Editor's Note: Timothy Stryker is the developer of RPL and his company, "Samurai Software" markets RPL for PET and CBM computers.

BASIC, Beginners' All-Purpose Symbolic Instruction Code, is an excellent language in many respects. It is easy to learn and easy to use. It is very tolerant of user error, which makes the debugging of programs in BASIC a relatively simple matter. Although many different versions of BASIC exist, enough of its features have become standardized to make it reasonably transportable. A BASIC program written on a PET or CBM will generally run on other machines with minor modifications, and *vice versa*.

The main problem with BASIC is its speed. If you have ever tried to write a real-time game or process control program in BASIC, you have no doubt found that it bogs down very easily. PET/CBM BASIC is one of the fastest floating-point BASICs in existence, but there are still many applications for which this BASIC is too slow to be of value. BASIC also consumes prodigious amounts of memory, both for storing user programs and for storing the data to be processed during execution. This can lead to problems with the OUT OF MEMORY error in the course of writing large applications programs.

BASIC's lack of speed has caused many programmers to become interested in FORTH as a widespread language for small computers. Because FORTH usually manipulates numbers in integer form, it gains a significant speed advantage over BASIC (microprocessors can manipulate integers much more quickly than they can floating-point numbers). In addition, FORTH is a "compiled" language, which means faster performance in looking up variables in tables, finding destinations of control-flow branches, and the like. Primarily through the efforts of the "FORTH Interest Group" ("fig") in San Francisco, FORTH has become sufficiently well-known to make possible a reasonable degree of transportability between machines. Unfortunately, FORTH's "extensibility" (its ability to allow the user to add new constructs of all kinds to the language) has led to numerous substantially different flavors of FORTH on the market. Nonetheless, most "fig-FORTH" versions adhere fairly closely to the Interest Group's standards.

The main reason that FORTH has not caught on more strongly is that the language is considerably more difficult to use than BASIC. FORTH operations are ordered according to Reverse Polish Notation, which many people find objectionable. At the same time, the fig-FORTH text editor and its associated disk I/O standards are both unique and cumbersome, which makes FORTH source file management difficult and error-prone. The lack of worthwhile FORTH debugging tools has not helped the situation either: once debugged, FORTH programs tend to be fairly solid, but getting to that point can take a major effort.

In the midst of all this ferment, a new language called RPL has appeared on the scene. RPL, which stands for Reverse Polish Language, is to some extent a combination of BASIC and FORTH. RPL is a compiled language in the same sense that FORTH is. RPL object code is not itself machine code, but it can be interpreted by a machine-language "run-time executive" that is part of the package. RPL also uses both a parameter stack and a return stack, just like FORTH. However, RPL resembles BASIC in many respects: its implementation on the PET/CBM uses Commodore's BASIC screen editor and all of the normal BASIC source file manipulation commands, like SAVE and DSAVE, LOAD and DLOAD, LIST, DIRECTORY, and so on. This means that RPL programs can be of arbitrary length, without your having to break them up into 1024-byte sections the way you must with FORTH programs. Also, RPL program listings read from top to bottom, just like BASIC listings (which is to say, unlike FORTH listings).

Interestingly enough, RPL is substantially more efficient than FORTH in both space and time in spite of the fact that it is easier to use. Since dedicated FORTHers will no doubt find this hard to believe, I have assembled a few small benchmarks on the CBM 8032 in order to compare BASIC, fig-FORTH, and RPL in terms of their processing speeds and their memory usage.

The Block-Move Benchmark

Listings 1a, 1b, and 1c show BASIC, fig-FORTH, and RPL implementations of a simple block-move benchmark. The three versions have been kept in as close a correspondence to one another as possible — thus, line 150 of the BASIC version corresponds directly to line 150 of the RPL version and to line 6 of the FORTH version, and so on. Since the routines must appear in bottom-up order in the FORTH version, the symmetry is somewhat broken here, but the BASIC and RPL versions are line-for-line equivalents of one another.

Each benchmark begins by zeroing the 8032's internal timer so that timing measurements can be made. In BASIC, this is accomplished by setting the variable TI$ to a string of six zeroes, whereas in both FORTH and RPL, this

is done by storing a zero into the word at memory location 142 (the sequence {0 142 !} accomplishes this in both languages).

Then, a 100-pass loop is set up so that the routines to be tested will each be run 100 times. Here we notice the first difference between FORTH and RPL: the RPL version does a {100 1 FOR} to accomplish this, whereas the FORTH version does a {101 1 DO}. RPL's FOR is the equivalent of FORTH's DO, except that in FORTH the upper bound of an iterative loop like this must always be specified as 1 greater than the actual upper bound desired. RPL is more like BASIC in this regard, as you can see.

The body of the loop in each case consists of setting up parameters to be passed to the routine under test, followed by a call to the routine itself. In the BASIC version, setting up of the parameters is accomplished by assigning values to the variables C, T, and F, which in this case specify that the 150 bytes starting at 634 are to be moved up to start at 826. The FORTH and RPL versions of the benchmark, however, expect these arguments to be passed to them on the parameter stack.

A crucial difference between FORTH and RPL is apparent here in the way in which the call itself is done: note that, in FORTH, the simple statement {BLKM} is sufficient to invoke the routine of that name. In RPL, saying simply {BLKM} merely causes the *address* of the BLKM routine to be *pushed onto the stack*: it is the {&} operator (pronounced "call") that actually causes control to be transferred to the routine whose address appears on top of the stack. There are numerous reasons why calls are set up this way in RPL, and we do not have the space here to go into them in any detail. Suffice it to say that the reasons center around space efficiency, speed, and ease of use, all three of which are optimized through the use of this construct. As evidence of this I can only cite the results of the benchmarks given here. These results are almost wholly due to precisely this difference between FORTH and RPL: in FORTH, you call a routine by simply stating its name, whereas in RPL you call a routine by stating its name and then invoking the {&} ("call") operator.

Listing 1A

```
100 REM ************************************************
110 REM *          BASIC BLOCK-MOVE BENCHMARK          *
120 REM * ROUTINE AT LINE 1000 MOVES C BYTES FROM F TO T *
130 REM ************************************************
140 REM
150 TI$="000000" : FOR I = 1 TO 100
160 C=150 : T=826 : F=634 : GOSUB 1000
170 NEXT I : PRINT TI;"JIFFIES" : END
1000 M=F-T : FOR J = T TO T+C-1 : POKE J,PEEK(J+M) : NEXT : RETURN
```

Listing 1B

```
0 ( ********************************************************** )
1 ( *                FORTH BLOCK-MOVE BENCHMARK              * )
2 ( * BLKM EXPECTS FROM-ADDR ON TOS, THEN TO-ADDR, THEN COUNT * )
3 ( ********************************************************** )
4
5 : BLKM OVER - SWAP ROT OVER + SWAP DO DUP I + C@ I C! LOOP DROP ;
6 : TEST 0 142 ! 101 1 DO
7   150 826 634 BLKM
8   LOOP 142 @ INT . ." JIFFIES" ;
```

Listing 1C

```
100 ************************************************************************
110 *                   RPL BLOCK-MOVE BENCHMARK                           *
120 * THE BLKM ROUTINE EXPECTS FROM-ADDR ON TOS, THEN TO-ADDR, THEN COUNT *
130 ************************************************************************
140 REM
150 0 142 ! 100 1 FOR
160 150 826 634 BLKM &
170 NEXT 142 @ INT STR$ PRINT " JIFFIES" PRINT STOP
1000 BLKM: ; - % 3 $ ; + 1 - % FOR # FN + PEEK FN POKE NEXT . RETURN
```

Table 1 shows a few of the RPL operators and their FORTH equivalents. Using this table, you can see that the FORTH and RPL versions of the BLKM routine itself are virtually word-for-word equivalents of one another, the only difference being that, as before, the upper bound of the DO-LOOP in FORTH is 1 greater than the upper bound of the FOR-NEXT in RPL and BASIC. Note that FORTH's {ROT} operator is equivalent to the sequence {3 $} in RPL: RPL's {$} operator takes the item on top of the stack and uses it to determine how deep into the stack the rotation process will go.

(Continued on page 66)

***Table 1:* A few of the RPL operators and their FORTH equivalents (TOS means Top Of Stack; NOS means Next On Stack).**

RPL	FORTH	Effects
+	+	Add TOS to NOS, pop TOS
–	–	Subtract TOS from NOS, pop TOS
\	MOD	Take NOS *modulo* TOS, pop TOS
IF	IF	Begin conditional based on TOS
THEN	ELSE	End THEN-part, begin ELSE-part
END	THEN	End conditional
@	@	Replace TOS with the word it points to
!	!	Store NOS into word pointed to by TOS, pop both
PEEK	C@	Replace TOS with the byte it points to
POKE	C!	Store NOS into byte pointed to by TOS, pop both
#	DUP	Push a new copy of TOS
;	OVER	Push a new copy of NOS
.	DROP	Drop (or pop) TOS
%	SWAP	Swap TOS and NOS
$		Rotate out TOS'th stack entry onto TOS

Once the block-move routine has been executed 100 times, each program then prints out the number of "jiffies" (1/60ths of a second) that have elapsed since it started up. In BASIC, this consists of simply printing out the variable TI. In FORTH and RPL, the contents of location 142 are fetched onto the stack, "byte-interchanged", and printed out. The "byte-interchange" operation {INT} is necessary only because PET/CBM BASIC stores the timer in high-order-byte-first order, whereas both FORTH and RPL expect fetched quantities to appear in the usual low-order-byte-first order in memory.

In the process of printing out the jiffy count and the word JIFFIES, we see another fundamental difference between FORTH and RPL: RPL treats character strings as an elementary data type, whereas standard FORTH does not. The FORTH {.} operator both converts the top stack entry to ASCII and prints it out, and FORTH's {."} operator unconditionally prints out the character string following it — at no time does fig-FORTH leave a character string sitting on the stack in such a way that the user can get at it. In RPL, character strings representing numbers are frequently placed onto the stack in such a way that the user can then manipulate them to any desired purpose, because RPL's {STR$} operator merely converts the top stack entry to a character string, and the PRINT operator is necessary actually to print the number out. Similarly, when RPL comes across a literal character string enclosed in quotes, it simply pushes that string onto the stack: once the string is on the stack, it may be manipulated further, or, as in this case, immediately printed out using PRINT.

Table 2: **Results from the Block-Move Benchmark**

	Program Bytes	Data Bytes	Jiffies	Figure of Merit
BASIC	115	42	6044	23.23
FORTH	92	0	591	1.82
RPL	57	0	525	1.00

Table 2 displays the results from this first benchmark. In BASIC's case, the "Program Bytes" column does not count REMarks or spaces, but only the actual amount of memory taken up by the code itself. Many programs are available that compress BASIC code by removing all REMark lines and extraneous blanks, and the figure given here applies to the code size following a compression of this sort. In FORTH and RPL, the size of the object code is of course independent of the number of comments and spaces appearing in the source.

The FORTH and RPL "Program Bytes" entries pertain only to the object code actually generated by the portions of the programs shown in listings 1b and 1c. It should be kept in mind that both of these languages actually incur about one additional K in minimum run-time memory overhead — in FORTH's case, for the so-called "inner interpreter;" in RPL's, for the so-called

Listing 2A

```
100 REM ************************************************************
110 REM *                    BASIC SHUFFLER BENCHMARK                *
120 REM * THE ROUTINE AT LINE 1000 RETURNS A SHUFFLED DECK IN A *
130 REM ************************************************************
140 REM
145 DIM A(51)
150 TI$="000000" : FOR I = 1 TO 100
160 GOSUB 1000
170 NEXT I : PRINT TI;"JIFFIES" : END
1000 FOR J = 0 TO 51 : A(J)=J : NEXT
1010 FOR J = 1 TO 200
1020   R1=INT(52*RND(1)) : R2=INT(52*RND(1))
1030   T=A(R1) : A(R1)=A(R2) : A(R2)=T
1040 NEXT : RETURN
```

Listing 2B

```
 0 ( ************************************************** )
 1 ( *              FORTH SHUFFLER BENCHMARK          * )
 2 ( * THE SHUFFLE ROUTINE RETURNS A SHUFFLED DECK IN DECK * )
 3 ( ************************************************** )
 4
 5 0 VARIABLE DECK 52 ALLOT
 6 : SHUFFLE 52 0 DO I DECK I + C! LOOP
 7   201 1 DO
 8     52 RND 52 RND
 9     DECK + DUP C@ SWAP ROT DECK + SWAP OVER C@ SWAP C! C!
10   LOOP ;
11 : TEST 0 142 ! 101 1 DO
12   SHUFFLE
13   LOOP 142 @ INT . ." JIFFIES" ;
```

Listing 2C

```
100 ************************************************
110 *                RPL SHUFFLER BENCHMARK          *
120 * THE SHUFFLE ROUTINE RETURNS A SHUFFLED DECK IN DECK *
130 ************************************************
140 REM
150 0 142 ! 100 1 FOR
160 SHUFFLE &
170 NEXT 142 @ INT STR$ PRINT " JIFFIES" PRINT STOP
1000 SHUFFLE: 51 0 FOR FN DECK FN + POKE NEXT
1010 200 1 FOR
1020   RND 52 \ RND 52 \
1030   DECK + @ PEEK % 3 $ DECK + % ; PEEK % POKE POKE
1040 NEXT RETURN
1050 DECK:(<52>)
```

"run-time executive." I would have included this memory overhead in a separate column if not for the fact that standard fig-FORTH makes it very difficult to separate out the inner interpreter from the rest of the FORTH operating system. Many FORTH packages include some sort of special "target" compiler program which facilitates this process (a capability which is an inherent part of the RPL package, by the way), but mine is not one of them. In any event, because this fixed overhead for FORTH and RPL has little to do with the relative space-efficiencies of the two languages, it was ignored for the purpose of doing these benchmarks.

BASIC users may be perplexed by the figure of zero "Data Bytes" given in this benchmark for both FORTH and RPL. Naturally, both FORTH and RPL do manipulate data, and this data does need to be stored someplace during execution. In this case, though, all of the storage needed for both of these versions exists on their stacks. This makes the data storage essentially "free of charge." If this bothers you at all, consider that, unbeknownst to you, BASIC also has a stack that it uses extensively for various purposes, and the stack space that BASIC uses in these benchmarks has not been counted against it in the "Data Bytes" figures either.

The column entitled "Figure of Merit" in table 2 is based around the notion that the overall efficiency of a language is a function of both its time-efficiency (speed) and its space-efficiency. One fairly common way to combine these two measures of efficiency is to multiply each version's program size by the amount of time it took to execute: the lower this number is, then, the more efficiently the language handled the benchmark. In table 2 these Figures of Merit have been normalized in order to show their ratios to RPL.

The Shuffler Benchmark

Listing 2 shows BASIC, FORTH, and RPL versions of a benchmark designed to test the languages' efficiencies at a typical game-related task. Each version times 100 passes through a "card shuffling" routine, in which array entries of 0 through 51 are used to represent the 52 cards of a normal playing deck.

Line 145 of the BASIC version, line 5 of the FORTH version, and line 1050 of the RPL version all accomplish essentially the same thing — each allocates space for a 52-element array that will be used to store the simulated deck of cards. This brings out another fundamental difference between RPL and FORTH: in RPL all handling of user-defined symbols is carried out in just one way, whereas FORTH handles user-defined symbols differently depending on whether they are subroutine names, variable names, constants, etc. RPL's symbol-handling concept is similar in many respects to that used in assembly language, in that a given user-defined symbol in RPL provides merely an alternate method of specifying what would otherwise appear in the code as a literal numeric value or address. As anyone who has programmed in assembly language knows, this is a simple, but surprisingly powerful way of handling symbols: in particular, the ability to treat addresses of routines as ordinary data can be very useful in certain applications. None of the benchmarks shown here makes use of this capability, because neither FORTH nor BASIC has a corresponding capacity along these lines. Note, however, that RPL makes no real distinction between the symbol DECK defined in line 1050 of listing 2c and the symbol SHUFFLE defined in line 1000. Each of them simply takes on a value equal to the address of the subsequent byte of object code, and, when referred to, simply causes that address to be pushed onto the parameter stack at run-time.

Each of the three versions of the shuffling routine operates in basically the same way: the 52-element array is initialized to the numbers 0 through 51, in sequence, and the order of these elements is then randomized by means of swapping random pairs a total of 200 times. The results of this benchmark are shown in table 3. The astonishingly long time taken by FORTH in this benchmark seems to be largely due to its handling of the MOD function, the invocation of which is internal to my version of the FORTH RND routine. The MOD operator under fig-FORTH on the CBM 8032 takes over 4.5 milliseconds to execute, whereas its RPL equivalent, { \ }, takes less than 1.2 milliseconds, worst case. Since the two

operators yield identical results, it should be recognized that FORTH's poor performance in this benchmark is not primarily a function of anything inherent in the language itself, but is largely due to the speed of the modulo algorithm chosen by the FORTH Interest Group.

The Falling-Tone Benchmark

Listing 3 contains BASIC, FORTH, and RPL versions of a benchmark designed to test the languages in terms of general logical and arithmetic manipulations, including comparisons, conditional branching, and memory accesses. Each program times itself doing 100 calls to a routine that generates a falling whistle on the 8032's internal speaker. The method used to generate the falling whistle is based on the "VDC" algorithm (see the 10/81 issue of *BYTE*, p. 391). Each octave drop in pitch takes the same amount of time, regardless of whether the octave is toward the top of the range or toward the bottom (the same cannot be said of the obvious "FOR I=1 TO 255:POKE 59466,I:NEXT" in BASIC).

This benchmark brings out yet another major difference between RPL and FORTH. First of all, note that in the BASIC version of the benchmark, a "conditional-within-a-conditional" (in line 1030) takes control out of the loop in lines 1020-1050 if the condition is met. There is every reason to suppose that this is a perfectly "structured" thing to do: only if the first condition (in line 1020) is not met will we determine whether or not it is time to exit the loop. FORTH, however, does not permit this kind of construct. One may set up a BEGIN...WHILE...REPEAT loop in FORTH, but the WHILE operator may not appear within the bounds defined by an IF...THEN pair within the loop. This is restrictive, to say the least, and in an application like this one it unavoidably leads to slower code. The best I could do to get around this in FORTH was to place the WHILE test *outside* the main conditional clause, which meant that it got executed on *every* pass through the loop, regardless of whether or not it needed to be. RPL, being much more like BASIC in this regard, is able to get around this problem through the use of a GOTO (horrors!). This naturally opens Pandora's Box as far as hard-core structured-programming people are concerned. Suffice it to say that I feel that the real value of structured programming lies in its concern with modularization and clean, well-thought-out software design, not in terms of myopic, over-applied dogmas such as "No GOTOs!" and "No Multiple Entry Points!", etc.

Table 4 shows the results from this third benchmark. It should be clear from these various figures that FORTH is more efficient than BASIC at handling the kinds of tasks shown here, and that RPL is even more efficient than FORTH at these types of tasks, by perhaps a factor of two overall.

Other Tradeoffs

Ease of use is a very important criterion in determining the real utility of a language. BASIC is the acknowledged leader in this area, at least as far as "quick-and-dirty" programming is concerned. One of BASIC's best qualities is the interactive nature of its pro-

gram debugging facilities: the BASIC programmer has the option of halting his program at any point so that he can examine variables of interest, etc., and execution can then be continued from where it left off. FORTH has similar debugging tools, but their utility is blunted somewhat due to the opaque nature of the FORTH stacks, not to mention the clumsiness of FORTH's editing, source file handling, and so on. RPL solves this problem by providing a "symbolic debugger" as a separate utility program that allows any RPL program to be debugged *via* single-stepping, breakpointing, and the like. And the entire contents (up to 18 entries deep) of both the parameter stack and the return stack are available for viewing at any time.

"Extensibility," or the ability of the language to be augmented by the user, is one area in which FORTH shines brightly. Two distinct capabilities of FORTH here are sometimes lumped together in reviews of the subject. One is that normal FORTH routines, once defined, become as though part of the language itself. This feature is really no different in principle from the conventional method of subroutine calling used in other languages. The other capability is much more intriguing: the {<BUILDS} and {DOES>} operators give the FORTH user the ability to effectively modify the FORTH compiler on the fly, so that whole new language constructs can be created. No other language I know of (with the possible exception of Ada) has this feature. How desirable this is, to many people, remains to be seen: the benefits accruing to it must certainly be balanced against the code obscurity and tendency toward destandardization resulting from it. For example, the same capability is implemented in some FORTH versions as BEGIN...IF...WHILE, in some as WHILE...PERFORM...PEND, and in some as BEGIN...WHILE...REPEAT.

It is in the area of transportability that BASIC and FORTH find their strongest advantage over RPL. RPL is presently available only for the Commodore PET and CBM series of machines, whereas BASIC and FORTH have both become widespread. Every new language, though, goes through a period of limited transportability in its early stages. If the language really does present worthwhile advantages over existing languages, it will eventually be adapted to run on systems other than the one on which it was developed. In fact, Samurai Software is now actively seeking individuals who would be interested in adapting RPL to the Apple, the TRS-80, CP/M, and so on. Would *you*, by any chance, be interested?

Listing 3A

```
100 REM ************************************************************************
110 REM *                    BASIC FALLING-TONE BENCHMARK                      *
120 REM * ROUTINE AT LINE 1000 GENERATES CB2 TONE WITH EXPONENTIAL FALLOFF *
130 REM ************************************************************************
140 REM
150 TI$="000000" : FOR I = 1 TO 100
160 GOSUB 1000
170 NEXT I : PRINT TI;"JIFFIES" : END
1000 POKE 59464,0 : POKE 59467,16 : POKE 59466,170
1010 DY=20 : DC=0
1020 IF DC>=0 THEN 1050
1030 DY=DY+1 : IF DY=256 THEN 1060
1040 DC=DC+256 : POKE 59464,DY
1050 DC=DC-DY : GOTO 1020
1060 POKE 59467,0 : POKE 59466,0 : RETURN
```

Listing 3B

```
 0 ( ************************************************************** )
 1 ( *                  FORTH FALLING-TONE BENCHMARK                * )
 2 ( * THE TONE ROUTINE GENERATES CB2 TONE WITH EXPONENTIAL FALLOFF * )
 3 ( ************************************************************** )
 4
 5 : TONE 0 59464 C! 16 59467 C! 170 59466 C!
 6   20 0
 7   BEGIN DUP 0< IF
 8   SWAP 1+
 9   SWAP 256 + OVER 59464 C! THEN OVER 256 < WHILE
10   OVER - REPEAT
11   DROP DROP 0 59467 C! 0 59466 C! ;
12 : TEST 0 142 ! 101 1 DO
13   TONE
14   LOOP 142 @ INT . ." JIFFIES" ;
```

Listing 3C

```
100 ************************************************************************
110 *                    RPL FALLING-TONE BENCHMARK                        *
120 * ROUTINE AT LINE 1000 GENERATES CB2 TONE WITH EXPONENTIAL FALLOFF *
130 ************************************************************************
140 REM
150 0 142 ! 100 1 FOR
160 TONE &
170 NEXT 142 @ INT STR$ PRINT " JIFFIES" PRINT STOP
1000 TONE: 0 59464 POKE 16 59467 POKE 170 59466 POKE
1010 20 0
1020 LOOP: # 0 < IF
1030 % 1 + # 256 = IF . . THATSIT GOTO END
1040 % 256 + ; 59464 POKE END
1050 ; - LOOP GOTO
1060 THATSIT: 0 59467 POKE 0 59466 POKE RETURN
```

Table 3: Results from the Shuffler Benchmark

	Program Bytes	Data Bytes	Jiffies	Figure of Merit
BASIC	179	367	48175	23.15
FORTH	117	54	15136	4.75
RPL	70	52	5321	1.00

***Table 4:* Results from the Falling-Tone Benchmark**

	Program Bytes	Data Bytes	Jiffies	Figure of Merit
BASIC	219	21	63701	32.54
FORTH	150	0	5764	2.02
RPL	96	0	4466	1.00

Timothy Stryker may be contacted at Samurai Software, P.O. Box 2902, Pompano Beach, FL 33062.