Behavior Research Methods, Instruments, & Computers
1986, 18 (2), 228-235

FORTH, a software solution to real-time computing problems

BRIAN H. WATTS
New York University, New York, New York

Some common real-time computing problems are discussed, namely, the programming and testing of devices, problems relating to program execution speed, the timing of experimental events, and the programming of multiple parallel events. The programming language FORTH is shown to be capable of providing solutions to all of these problems. In addition, FORTH has the advantages of being an interpretive and extensible language with both high-level and low-level capabilities, while at the same time being efficient and fast in execution.

In the first part of this article, six major problems encountered in the programming of real-time experiments are discussed. This is followed by an outline of some traditional ways of solving these problems, using methods based on hardware, firmware, and software, and an indication of the extent to which these methods are found lacking. After a brief introduction to the programming language FORTH, a description is given of how FORTH can overcome these problems. The elegance of FORTH program code is illustrated by means of a sample experiment and the use of FORTH in the psychology laboratory is evaluated.

SOME COMMON PROBLEMS

Real-time computer programming is fraught with logistical problems. Some problems arise because the computer used has insufficient power, and others are a result of the machine's being too powerful. The second statement may seem to contradict the first, but it is often the case that powerful computers have inherent complexities which, although normally transparent to the user, become significant when such computers are used in a real-time environment. For this reason, an efficiently programmed microcomputer may be far more suitable than many alternative minicomputers for running an experiment. Nevertheless, programming microcomputers for experiments is not without difficulties. Some of the most common real-time programming problems are the following.

Speed

In real-time applications, it is frequently important to optimize program execution speed because there are strong time constraints for the setting up of stimuli, the collection of data, and so forth. Execution speed can be retarded by the software, the hardware, and the data communication channels.

Software. Most computer languages can be categorized as low-level or assembler languages, or as high-level languages, which may be either compiled or interpreted. Some high-level languages can make use of certain low-level features, such as bitwise Boolean arithmetic, and some can be linked to assembler routines. The advantages of low-level languages are flexibility and speed, whereas high-level languages are more problem oriented and are easier to maintain and debug. Interpretive languages usually sacrifice speed for a dramatic reduction in program development time.

Hardware. The slow speed of the microprocessor of a microcomputer can be a prohibitive factor for certain psychological experiments. For example, the generation of rapidly updated complex graphic displays or the collection of electroencephalographic (EEG) or eye-movement data may require the speed of a minicomputer, or at least a high-powered state-of-the-art microprocessor, such as the Motorola 68000. Since the purpose of this article is to focus on software solutions to real-time programming problems, I will not further address the question of hardware limitations.

Data communication channels. A major problem with microcomputers in the laboratory is the limited access time, data transfer rate, and storage capacity of secondary storage devices, which normally take the form of floppy disks. Experimental trials may, for example, have to be interrupted by periodic data transfer and/or disk swaps. Hard disks improve the situation, but they are relatively expensive.

Device Communication

Absolute memory access. A problem faced by many high-level programmers of microcomputers is that of "talking" to peripheral devices. In the case of memory-mapped I/O, this is accomplished by referencing an absolute memory location. It is difficult to do this in most

I thank Doris Aaronson and John K. Adams for their helpful comments on earlier drafts of this paper.

The author's mailing address is: Department of Psychology, New York University, 6 Washington Place, 8th floor, New York, NY 10003.

high-level languages, because high-level languages normally "protect" the user from crashing the computer by writing to illegal memory locations. This protection mechanism is implemented by making absolute address references difficult, or even impossible. Although ROM (read only memory)-based interpreters, such as the BASICs resident on most microcomputers, are automatically protected and thus usually allow absolute memory addressing, they have other possibly more serious limitations.

Bit operations. Through a sequence of multiplications, divisions, and comparisons, bitwise Boolean operations can be simulated in high-level languages, but the time required to perform these operations is likely to exceed the limits imposed by the real-time device/port servicing constraints. For this reason, it may be necessary to perform these operations within a machine-code routine (the mere mention of which brings an instant shudder to many an unseasoned programmer).

Device Testing

The fact that devices are notoriously unreliable introduces a level of complexity in the development of real-time software not seen in normal applications. If one has a number of devices, such as a voice key, response key, physiology monitoring equipment, and so forth, connected to a computer, the possibility that one or more of these devices will malfunction is relatively high. The problem may be a simple one, due, for example, to the machine's not having been turned on or to its having been incorrectly calibrated. The symptoms of such a malfunction may be unpredictable, especially when a new (or modified) program is being tested. For this reason, it is essential to be able to test out devices, and to troubleshoot faulty device/program combinations independently and, preferably, interactively. Unfortunately, it is usually extremely difficult to do this when the program is written in a compiled language, such as FORTRAN or even PASCAL.

Timing

Reaction time recording. In experiments in which reaction times are recorded, these data may be obtained by means of a hardware real-time clock, a programmable timer, or a software clock. Although it has the disadvantage of not being able to handle the timing of parallel events easily, and although the processor may be required for other tasks while the timed event is taking place, the software clock can be easily implemented and is adequate for most experimental situations. In spite of this, the accurate recording of reaction times is often a major problem in the programming of psychological experiments.

Programming of fixed real-time delays. The options available here are the same as those available for reaction time recording, but fixed time delays typically require lower resolution. For example, although it may be reasonable to use a software delay timer that has a resolution of 100 msec, accurate to within 10 msec, this degree of tolerance may not be acceptable when measuring reaction times.

Multitasking

Multitasking is the programming of parallel or pseudo-parallel events, which may or may not communicate with one another. The majority of microcomputers do not have the hardware to enable them to perform multitasking, since multitasking systems usually require multiple hardware stacks, multiple interrupt levels, and so forth. When collecting data simultaneously from a number of subjects, multitasking may be the only feasible programming alternative. Running multiple subjects, however, is not the only use of multitasking in the laboratory. Consider an experiment in which eye movements are monitored while the computer generates a complex graphics display. In order to record the eye-movement data, the graphics routines need to be interrupted at various points. One way of doing this without using multitasking is to have a hardware clock generate interrupts at fixed real-time intervals and an interrupt handler record the information. Problems arise, however, when the interrupt handler itself needs to be interrupted, requiring multiple interrupt levels and complex programming. Another approach is to set up the two procedures (eye-movement recording and graphics display), as parallel tasks within a multitasking system. The task scheduler sees to it that the tasks are run in parallel (actually, they are run sequentially, but are switched very rapidly, mimicking parallelism), and the programmer is freed from the burden of having to synchronize processes. Thus, if multitasking is available, it can considerably reduce the programming effort required.

Implementation

The programming, debugging, and maintenance of software to run an experiment can be more time-consuming than any other part of the experiment. This is particularly true in a real-time environment in which programs that run experiments can be very fragile and difficult to test. An interactive programming language makes testing and debugging much easier. The problem is that there are not many languages that are interactive, that support modular programming, and that have the speed and power of compiled languages. One alternative is to use interpretive BASIC for program development, using a BASIC compiler to compile the finished product. Unfortunately, the compiled versions very often behave differently from their interpreted counterparts. The differences become significant in a real-time application, especially when procedures such as software timers are employed. Another problem with compiling BASIC is that it is difficult to combine assembler and BASIC code—parameters are not easily passed, the BASIC program can be overwritten, the assembler routines cannot be tested easily, and so forth. In addition, BASIC has many problems of its own, not the least being that it has limited power.

SOME SOLUTIONS

Before proceeding to a discussion of how FORTH enables the programmer to overcome these problems, I will outline a few alternative hardware, firmware, and software approaches.

Hardware—VAX 11/780

Polson, Miller, and Karat (1981) describe a real-time system that runs on a VAX 11/780 under the VMS operating system. They report that what makes the VAX so powerful in a real-time environment is its hierarchical system of 32 priority levels, and its ability to multitask concurrent processes efficiently, switching context in 180 μsec. This "brute force" method is fine if one has a VAX 11/780 to spare for running experiments, but alas, this is not usually the case. In addition, because of the layers of the operating system that one has to deal with in order to communicate with devices, the interactive testing of devices may be far more difficult on a minicomputer than on a microcomputer.

Hardware/Firmware: Dedicated Systems

The microcomputer interface described by Ratcliff and Layton (1981) is an example of a dedicated system with its own command language burnt into its resident firmware. The system is located between the host computer and a terminal and controls terminal and device I/O. Experiments are programmed on the host computer, which sends sequences of high-level commands to the interface. The idea is an elegant one, but suffers from problems of portability, hardware cost, and lack of flexibility. Moreover, it does not handle multitasking.

Software

BASIC/assembler combinations. Adams (1985) describes a method for the accurate timing and rapid presentation of verbal stimuli. The system works well, but is heavily dependent on assembler routines and requires a complex interaction among BASIC, the assembled routines, and the disk operating system. Orchestrating the interaction of these three systems is not easy. Moreover, the system is designed for a particular class of experiments and cannot be easily modified to include experiments involving graphics as opposed to ASCII text displays.

Command interpreters. The function of command interpreters is to reduce program complexity, and hence development time, by providing the programmer with a software interface that is close to 100% problem oriented. That means, for example, that instead of having to worry about "POKEing" memory locations and fiddling with bits, the programmer concerns him/herself with such tasks as turning on lights, ringing bells, and so forth. These tasks are performed directly by means of (possibly parameterized) commands. An example is the SIMPLE command interpreter (Aaronson, Sherak, & Marcovicci, 1983), which runs under the UNIX operating system. The main problem with command interpreters is that although they simplify the control of the computer, by employing a reduced "instruction set" they also limit what can be done on the machine. For this reason, it is quite probable that a particular command interpreter will not be suitable for all the experiments one would like to perform.

THE FORTH SOLUTION

Before discussing the FORTH solutions to the real-time programming problems mentioned above, I will briefly introduce the reader to the FORTH programming language. A more detailed introduction is provided by "Starting FORTH" (Brodie, 1981), a beginner's level text; "Thinking FORTH" (Brodie, 1984) is recommended for the advanced programmer.

FORTH is a programming language developed over a number of years, beginning in the late '60's, by an astronomer, Charles Moore, for the purpose of controlling observatory telescopes in real time. The name stems from the word *fourth*, which was chosen because FORTH was considered a fourth-generation computer language. The *u* was dropped because on the IBM 1130, on which FORTH was implemented, file names were restricted to five letters. Because of its extremely efficient use of memory, FORTH has been implemented on virtually every type of microcomputer that exists, and is becoming increasingly popular as microcomputers become more widely used.

In FORTH, all procedures are called *words*. Words are separated from each other by spaces and can consist of any sequence of characters, including nonprinting ones. The words communicate with one another via a pushdown stack—words that require parameters pick them up off the stack, and words that generate results place these results on the stack. FORTH is the most popular of a family of languages known as threaded interpretive languages. These languages have an inner interpreter and an outer interpreter. The outer, or text, interpreter executes commands from the keyboard and compiles FORTH words into a compact executable form. The inner, or address, interpreter is continually running when a program executes. It traces its way through word definitions, linking to lower subdefinitions and popping back up to calling definitions.

FORTH is both fast in execution and compact. It is fast because although it is interpretive its inner interpreter is typically coded in about 10 machine instructions. Its compactness stems from the fact that procedure calls use only 2 bytes, and FORTH encourages modular programming with very little repetition of code.

FORTH is more like a virtual computer than a programming language. It has two stacks: a computation stack and a return stack. It also has a number of software registers and primitive instructions which operate on the stacks and are similar to hardware instructions.

FORTH is written in FORTH, and every system instruction is available to the programmer. To define a FORTH word, one uses the words ":" (start definition) and ";" (end definition). The name of the new word immediately follows the colon. For example, typing the sequence:

```
: SQUARED DUP * ;
```

causes a new word (SQUARED) to be compiled into the FORTH dictionary. This word, when invoked, will duplicate the number on top of the stack (DUP), then multiply the top two stack items, replacing them with the result of the operation (*). After having compiled this definition, if the user were to type

```
4 SQUARED.
```

the number 16 would appear on the screen (the word "." prints the number on top of the stack). From then on, one can use the word SQUARED within another definition or type it on the keyboard for immediate execution—a new command would have been added to the language.

Some people insist that FORTH is not really a computer language, but a collection of software tools. The source of this slightly misleading notion is probably rooted in the fact that, in FORTH, syntactical constraints are minimal. There are only a half dozen or so syntax errors that can be made in FORTH. Although it could be said that, because of this, the language lacks structure, it is also true that its syntactic simplicity and consistency make the language easy to learn. The "software tools" concept is not without its grain of truth, however. This is, in fact, one of the language's major strengths. FORTH encourages users to build up a collection of commonly used procedures irrespective of size; FORTH words frequently consist of only two instructions. These words are compiled into the FORTH dictionary and from then on are treated as though they were part of FORTH itself. In other words, FORTH is extensible. Enhanced FORTH systems can be saved onto disk so that when the system is next booted, FORTH will remember the previously defined extensions. To remove these extensions, one merely types "FORGET" followed by the name of the first extension word defined. It is thus extremely easy to tailor a FORTH system to one's normal programming environment. FORTH systems normally consist of an integrated package, which includes an assembler, a text editor, a disk operating system, optional utilities such as a file system, and the FORTH kernel. The strengths and weaknesses of FORTH will not be further discussed here. For this, the reader is referred to Lea's (1982) evaluation of FORTH as a real-time programming system. The rest of this paper will concentrate on discussing what FORTH can offer as a solution to the common real-time programming problems listed above, illustrating FORTH's power with an example of an experiment that was programmed entirely in FORTH.

Dealing With the Above Problems

Speed. For an interpreter, FORTH is fast. It is about 10 times faster than interpreted BASIC, and is comparable in speed to most compiled languages. In addition, because low-level machine instructions are so easily incorporated into FORTH programs, and given that programs typically run less than 10% of their code for more than 80% of the time (Brodie, 1984), the assembler coding of these "busy" routines enables FORTH programs to run at maximum speed when required. FORTH employs a sophisticated buffering system for disk I/O which reduces disk accessing to a minimum. There is no complicated disk file system in FORTH. Instead, the disk is divided into blocks of 1K bytes which are read into memory by the command "n BLOCK", where *n* is the desired block number. This simplicity makes FORTH I/O extremely fast.

Device communication. Because FORTH naturally allows the use of absolute memory address referencing, memory-mapped I/O can be performed easily. In addition, FORTH's built-in assembler automatically links together assembler code and high-level code, enabling time-critical routines to be coded in machine code with minimum effort. High-level FORTH has bitwise Boolean operations such as "AND," "OR," and "XOR" (exclusive or), enabling the testing and setting of bits to be performed without resorting to assembler code. Thus, in many cases, it is not necessary to consider low-level programming to communicate with devices.

Device testing. FORTH's modularity and interactivity enable the programmer to test out devices very easily. I will give an example of this in the next section.

Timing. As mentioned above, the timing of fixed delays and reaction times can be achieved with software or by means of a hardware clock. If at all possible, the software alternative should be chosen. It makes for a much more portable system and reduces expenses. For the two timing requirements, two separate assembler routines should be coded. Assembler coding is needed because, in many instances, timing resolution of millisecond accuracy is required, and, on an 8-bit microcomputer, unless it is run at maximum machine speed, it is not possible to increment a 32-bit counter and check all ports within 1 msec. The reaction time routine cycles around a loop which increments a counter and exits when the appropriate response is made, returning with the final counter value, which is then normalized to provide a real-time value.

The second routine sets up a counter with a fixed value, which ticks down within a loop and exits when the counter reaches zero. Before the routine is called, the appropriate value is set up by dividing the number of milliseconds' delay required by a suitable constant. This constant and the normalization constant are determined by counting the number of machine cycles within each iteration of the routine and multiplying this figure by the clock speed. The obtained value is the amount of real time it takes to execute a single tick of the software clock (TICK-TIME). For the fixed delay, the number of iterations required is calculated by the following equation:

$$\text{ITERATIONS} = \text{TIME} / \text{TICK-TIME},$$

where TIME is the delay time required and ITERATIONS is the countdown constant. For the reaction time clock, the same equation is used in a different form:

$$\text{TIME} = \text{ITERATIONS} \times \text{TICK-TIME},$$

where TIME is now the elapsed response time. Although the words to perform these operations must be written in low-level FORTH (assembler code), they do not have to be separately loaded and they are invoked in exactly the same way as high-level FORTH words are. In fact, the programmer need not even know whether or not previously defined words were coded in high- or low-level FORTH.

Multitasking. Although not available in all versions of FORTH, multitasking is easily implemented in FORTH. FORTH code is re-entrant, which means it can be used by many tasks simultaneously. Each task just requires its own computation and return stacks, plus a set of FORTH registers. Multitasking in FORTH is not generally implemented by means of interrupts; instead, each task decides when it is prepared to be interrupted, signaling this to the task scheduler, which uses a simple round-robin procedure for task activation. Thus, FORTH multitasking requires no special interrupt hardware. Task scheduling, however, increases the inner interpreter overhead quite significantly, and, in addition, the stacks can often be more efficiently implemented in a single task system. For example, the Apple has a zero-page address mode which makes page zero the ideal location for a single computation stack. The Apple also has a 256-byte hardware stack which can be used for the return stack. For these reasons, many microcomputer users prefer to work with a more efficient single-task system.

Implementation. FORTH is compiled, and it therefore enjoys the power and speed common to most other compiled languages. At the same time, FORTH is interactive, simplifying the testing and debugging phases of program development. Because all FORTH words are defined in terms of previously defined words, FORTH encourages top-down design and bottom-up coding, which speeds up program development time and results in more reliable programs. When time constraints require portions of code to be written in assembler, these routines are naturally and easily combined with other high-level routines— there is no special calling sequence, parameters are passed via the stack, and the routines can be tested interactively. Parameters are passed implicitly, reducing procedure-calling overhead, which encourages the coding of short, modular sections of code. FORTH's extensibility enables a tailored environment to be created, so that minimal programming effort is required once the language has been extended toward the common laboratory setting. This makes it possible to simplify the language enough for a person with little or no programming experience to use it in running an experiment. In other words, FORTH can be made to look like an application-specific command interpreter, while maintaining the full power of the complete language.

CUSTOMIZING FORTH

When getting involved with FORTH for the first time, it is useful to consider basic procedures that will be needed for writing programs within one's environment. These procedures should then be developed and fully tested in isolation before being incorporated into any specific application. The experiment to be described was run on an Apple II+ with 64K, a Videx 80-column card, and a Saturn 128K RAM card. The dialect of FORTH used was "FORTH T+," the author's own personal brand of FORTH, which is faithful to the FORTH 79 standard but has some significant enhancements. FORTH was not designed for any specific computer, and so does not have machine-specific features such as graphics utilities defined within the standard. Moreover, in keeping with the FORTH philosophy of compactness of code (the FORTH kernel takes up about 6K), the FORTH standard does not include string-handling procedures, floating-point arithmetic, arrays, or a file system. The other reason these features are left out of the standard is that they are very easily defined by the programmer and most users would prefer to use their own tailor-made add-ons rather than those defined by an arbitrary standard. It may, therefore, take a FORTH initiate a few months to get off the ground if he or she insists on developing these tools from scratch. There is not much sense in "reinventing the wheel" though, if these additions already exist. At this time, most of these extensions have been written for virtually all the computers on which FORTH is available and, since FORTH is public domain software, listings can be obtained at no, or very little, cost.[1] The following general-purpose software utilities have been developed.

Graphics

I wanted to run a number of experiments which involved the brief presentation of verbal stimuli followed by a mask. The experimental manipulations to be performed included varying orthographic features of the stimuli. To achieve maximum flexibility with respect to the physical characteristics of stimuli, I created a graphics system that enabled me to display text in graphics mode with a simple command called DISPLAY.

File System

Instead of employing a file system to access disk-based data, FORTH treats the disk as though it were an auxiliary directly addressable chunk of memory. To read a byte from the disk, all that is required is the block number and the byte offset. Disk reads and writes take place automatically. This feature makes it very easy to implement a virtual array capability in FORTH. In my system, except for initialization, virtual (disk resident) arrays appear to the programmer as though they were memory resident, making the storing of response data on disk a simple matter. If one has an external memory card, such as the Saturn 128K, the virtual memory capabilities become extremely powerful. Because it is a simple matter to make the Saturn 128K card look like a disk drive to the FORTH system, one can have virtual memory performing like main memory. Using this system, it is possible to transfer response data to, or load stimulus data from, the virtual disk with almost no delay.

Communications Program

The experimental stimulus words were obtained from the full Kučera and Francis (1967) word frequency dictionary on an IBM mainframe computer. To transfer these potential stimuli to the Apple, it was necessary to use a small communications program written in FORTH. The same communications program was used to transfer the experimental results from the Apple to a VAX 11/750 and an IBM mainframe for statistical analyses.

General-Purpose Timer

A general-purpose response timer (WAIT) was developed which could be used to monitor any of the three 1-bit input ports on the Apple. The address of the port to be examined is put on the stack prior to invoking the word. For example, the instruction

```
1 BUTTON WAIT
```

will cause the program to wait until pushbutton 1 is pressed, returning with the response time. A fixed-delay software timer (MSEC) was also implemented. The required delay in milliseconds is placed on the stack. Thus,

```
200 MSEC
```

will cause the program to delay for 200 msec.

Screen Vertical Blanking Synchronization

The electron beam of a standard display monitor takes 16.6 msec to get from the top to the bottom of the screen. This sweep of the beam is called the screen refresh, and is independent of the program being executed. This means that with this kind of monitor it is impossible to present a stimulus to a subject for less than 16.6 msec or, for that matter, for any nonintegral multiple of 16.6 msec. One cannot normally tell the beam where to go, but one can often find out where the beam is located at any instant. Many microcomputers can be programmed to determine when the beam is at the top of the screen by detecting the vertical blanking pulse. If one does not synchronize the setting up of stimuli with the vertical blanking pulse, there will be a variability of up to 16.6 msec in stimulus display times. A FORTH word, "PULSES," which halts the program until the specified number of blanking pulses occurs, was created. For example,

```
1 PULSES   STIMULUS DISPLAY
2 PULSES   MASK DISPLAY
```

will synchronize with the blanking pulse and will present the stimulus and maintain it for 33.2 msec (2 PULSES) before replacing it with a mask.

Random Number Generator

Two random-number generators were devised. The first one operates in high-level FORTH, and assumes the desired range to be on the stack before executing, returning a random number within the specified range. For example,

```
20 100 RND
```

will place a random number between 20 and 100 on the stack. This word was used to randomize the treatment condition within each experiment. The second random-number generator operates in machine code and returns a random byte (0-255). This was necessary to effect a rapid generation of stimulus masks on each trial.

Hardware Modifications

Two minor hardware modifications were made on the Apple II. The first modification was the connection of the vertical blanking pulse line to the 1-bit input port at memory address $C063 (see Reed, 1979, for a detailed description of this operation). This enabled stimulus displays to be synchronized with the screen refresh cycle. The second modification was the connection of the output line from a Grason-Stadler voice-operated relay to the 1-bit input port at $C062. By sampling the contents of this memory location, the FORTH word "WAIT" measured the subjects' vocal response times to within 1-msec accuracy.

AN EXAMPLE OF AN EXPERIMENT WRITTEN IN FORTH

I will now describe some of the FORTH code used in an experiment that examined effects of letter resolution and letter position for tachistoscopically displayed single-word stimuli. See Figure 1 for a graphic description of the experiment and the Appendix for a description of all FORTH words used in this article. In this experiment, there were three stimulus conditions: (1) the word was presented intact, (2) a random letter was deleted from the word, and (3) a random letter had half of its pixels randomly removed. There were four display durations, ranging from 1 to 4 pulses (16.6 to 66.4 msec). The trial sequence was randomized across all conditions. To enable the subject responses to be quickly and accurately recorded, they were typed in by the experimenter after the conclusion of each trial. For this experiment, it was important not to give the subjects accuracy feedback, so an independent display was needed for the experimenter's use (this was provided by the Videx 80 column card). I now describe each step of the experiment, together with the corresponding FORTH code (in capital letters).

1. STIM PRINT—The stimulus word is displayed on the experimenter's monitor.
2. FIXATION DISPLAY—The centered fixation point is displayed on the subject's monitor.

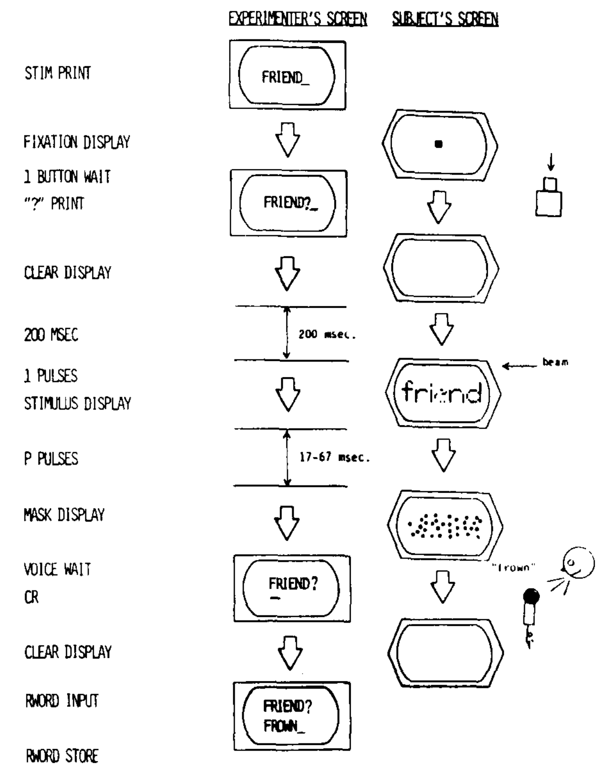

Figure 1. One trial of the tachistoscopic single word presentation experiment. Note that the letter "e" has had half of its pixels removed. This was the condition of "reduced resolution" in the experiment.

3. 1 BUTTON WAIT—The subject presses a button when ready for the trial.
4. "?" PRINT—The subject's buttonpress is signaled on the experimenter's screen.
5. CLEAR DISPLAY 200 MSEC—The subject's screen is cleared for 200 msec to avoid fixation point interference.
6. 1 PULSES STIMULUS DISPLAY—The stimulus is displayed after synchronization with the blanking pulse.
7. P PULSES—The stimulus remains displayed for 1-4 pulses (P is a variable that contains a number in the range 1 to 4).
8. MASK DISPLAY—The stimulus is covered by a random-dot mask.
9. VOICE WAIT—A software clock starts and waits for a response from the voice-actuated relay.
10. CR CLEAR DISPLAY—A vocal response causes the screen to be cleared. This event is signaled on the experimenter's screen by a carriage return.
11. RWORD INPUT—The experimenter enters the subject's response via the computer keyboard.
12. RWORD STORE—The subject's data are stored on disk in a virtual array.

In regard to the above code, it should be obvious that the FORTH code is extremely readable, self-documenting, and modular. Furthermore, once defined, each of the above words is added to the language, and can be individually and interactively tested. For example, the user merely needs to type in the command:

```
VOICE WAIT
```

in order to test the voice key. If it is working, the cursor will disappear and then reappear when the voice key is triggered.

CONCLUSION

The pros and cons of FORTH could be endlessly debated, although this approach misses the point that there is no perfect programming language, since every language has to compromise certain features to accommodate others. FORTH is no exception to this rule; it has many powerful features, but will be inadequate in many programming environments. As most assembler programmers know, there must be a tradeoff between the inherent power of a programming system and the protection or "user friendliness" it provides. Because program efficiency is so essential in a real-time programming environment, one is normally prepared to forego some protection in order to optimize performance. The reason why FORTH is so ideal for real-time programming is that it has just enough protection for the average programmer with almost no sacrifice in performance. Before rushing into a FORTH commitment, however, one should carefully balance the costs of learning and implementing a new system with the benefits that it might provide. To assist prospective FORTH initiates in making this decision, a few simple criteria are proposed. If any of the following circumstances apply to the reader, I suggest giving FORTH a try: (1) An interpreter is appealing, but interpretive BASIC is too slow. (2) Present capabilities do not provide sufficient internal memory to accommodate both the stimuli and the program to run experiments. (3) Similar types of experiments are being run, requiring the recoding of programs differing in small details. (4) The idea of a command interpreter that can be easily upgraded, and that is not limited by a restricted domain sounds inviting. (5) The user would like more control of the computer with fewer arbitrary restrictions imposed by the programming language.

On the negative side, FORTH can't do everything. There are no FORTH statistical packages which have the power of SAS, SPSS, or BMDP. But this isn't much of a problem, because nowadays a computer user is almost forced to distribute processing requirements across a number of different computers of various sizes. Because FORTH is unlike any other language (with the exception of FORTH clones and other threaded interpretive languages), it may initially be difficult to get used to. It is also true that FORTH gives one enough rope to hang oneself. The beginner might constantly "bomb out" the computer by overwriting part of the FORTH kernel. If researchers survive this possibly frustrating early part of the learning curve, I am confident that most of them will be pleasantly surprised at what FORTH has to offer in the psychology laboratory.

REFERENCES

AARONSON, D., SHERAK, R., & MARCOVICCI, S. (1983). *The SIMPLE users manual.* New York: New York University, Department of Psychology.

Adams, J. K. (1985). Visually presented verbal stimuli by assembly language on the Apple II computer. *Behavior Research Methods, Instruments, & Computers,* **17**, 489-502.

BRODIE, L. (1981). *Starting FORTH.* Englewood Cliffs, NJ: Prentice-Hall.

BRODIE, L. (1984). *Thinking FORTH.* Englewood Cliffs, NJ: Prentice-Hall.

KUČERA, H., & FRANCIS, W. (1967). *A computational analysis of present-day American English.* Providence, RI: Brown University Press.

LEA, D. (1982). Using the FORTH language in real-time computer applications. *Behavior Research Methods & Instrumentation,* **14**, 29-31.

POLSON, P. G., MILLER, J. R., & KARAT, J. (1981). Psychological experimentation through direct connection of the subject interface to a VAX. *Behavior Research Methods & Instrumentation,* **13**, 189-191.

RATCLIFF, R., & LAYTON, W. M. (1981). A microcomputer interface for control of real-time experiments in cognitive psychology. *Behavior Research Methods & Instrumentation,* **13**, 216-220.

REED, A. V. (1979). Microcomputer display timing: Problems and solutions. *Behavior Research Methods & Instrumentation,* **11**, 572-576.

NOTE

1. For more information, write: FORTH INTEREST GROUP, P.O. Box 1105, San Carlos, CA 94070.

APPENDIX
FORTH Words Used in This Article

The "Stack Before" column indicates parameters expected on the stack by the word. The "Stack After" column indicates parameters placed on the stack by the word. Numbers are indicated by the letter "n," and memory addresses by the letter "a." A "-" indicates nothing is expected or placed on the stack.

WORD	DESCRIPTION	STACK BEFORE	STACK AFTER
	FORTH Kernel Words Used in This Article		
:	Starts a word definition.	-	-
;	Ends a word definition.	-	-
DUP	Duplicates number on stack.	n	n n
"	Places the address of the following string delimited by " on the stack.	-	a
*	Multiplies the top two numbers on the stack, replacing them with the result.	n1 n2	n1*n2
CR	Prints a carriage return.	-	-
	Prints the number on the stack.	n	-
	Application-Specific Extension Words		
RND	Places a random number between n1 and n2 on the stack.	n1 n2	n
PULSES	Waits for n vertical blanking pulses.	n	-
WAIT	Waits until a response is detected at port a, and returns the response time n in milliseconds.	a	n
BUTTON	Places the address of the port specified by n on the stack.	n	a
VOICE	Places the address of the voice-key port on the stack.	-	a
MSEC	Waits for n milliseconds.	n	-
PRINT	Prints the text at address a.	a	-
DISPLAY	Displays the information at address a in graphics mode in the center of the screen.	a	-
INPUT	Accepts a text string from the keyboard, and places it at address a.	a	-
STORE	Stores the text at address a on disk.	a	-
FIXATION	Places the address of a buffer containing a fixation point on the stack.	-	a
CLEAR	Places the address of a buffer containing white space on the stack.	-	a
MASK	Places the address of a buffer containing a mask on the stack.	-	a
STIMULUS	Places the address of a buffer containing the stimulus on the stack.	-	a