Introduction to the Forth Programming Language

"I have yet to see a decent piece of software written in Forth. Let's face it. Forth stinks." --- John Dvorak, provocateur and columnist, InfoWorld, October 29, 1984.

"Forth is the first language which has been honed against the rock of experience before being cast into bronze." --- Charles Moore, inventor of Forth.

"Only brain-damaged programmers use Forth." --- Alan Holub, ex-C columnist, Dr. Dobb's Journal.

"Forth is like the T'ao; it is a Way, and is realized when followed. Its fragility is its strength; its simplicity is its direction." --- Michael Ham, ex-Forth columnist, Dr. Dobb's Journal.

As this charming collection of aphorisms demonstrates, Forth is not a programming language that provokes lukewarm, wishy-washy sentiments. With rare exceptions, Forth programmers are passionately enthusiastic about the language, and will bend your ear for hours about how it changed their lives. Most other programmers view it with the sort of distaste mingled with wary respect that is ordinarily reserved for Mafia chieftains and the authors of software viruses.How did Forth acquire this reputation as the "bad boy" of programming languages? Some of it is undoubtedly based on hearsay from pundits like Mr. Dvorak and Mr. Pournelle, on Rolling Stone interviews with scruffy-looking Northern California hackers, on well-meaning but misguided promotional efforts by Forth groupies, and on the chaotic appearing, uncommented Forth source code published in trade magazines (when any code is published at all). Some of it stems from sensational (and sensationalized) disasters like EasyWriter and Valdocs, both of which happened to be written in Forth. In addition, there is always the snobbery factor and the Not-Invented-Here syndrome; "if they didn't teach me about it at Stanford in Computing Science 1A, it can't possibly be any good."

But the largest ingredient in Forth's roguish renown seems to be an appalling ignorance of Forth's true nature and features in the general programming community. For example, Microsoft recently billed the use of threaded code in its QuickBasic product as a technological breakthrough, while exactly the same technique has been used in Forth systems for nearly twenty years. Reviewers marvel over the new integrated programming environments from Borland and Microsoft, oblivious to the fact that Forth was providing integrated environments with an interpreter, compiler, editor, and assembler in less than 16 KB of RAM for nearly 15 years before Turbo Pascal. Many of the authoritative reference works on programming languages don't mention Forth at all, and C programmers (of all people) are often heard criticizing Forth's "write-only" source code.

I find the contemptuous attitude toward Forth that is common among C programmers especially disconcerting, because the two languages are virtually contemporary, were designed with many of the same objectives, and require much the same sort of approach when solving a particular programming problem. This impression of similarity is supported, in my experience, by the ease with which Forth programs can be translated to C and vice versa. Applications written in Forth and C are also prone to many of the same sorts of bugs, because both languages are tightly coupled to the hardware and allow you to mangle pointers and data in any way you like.Forth is, like every other programming language, simply a tool with particular strengths and weaknesses that reflect the original concerns of its designer. In the appropriate settings, Forth is without peer, just as Fortran is inarguably the best tool for crunching numbers, LISP for crunching symbols, and Prolog for crunching rules. While it is not my intention to teach Forth programming in this section, I would like to place some of the claims and counterclaims about Forth in a more reasonable perspective.

Some Forth History

Although Forth and C were conceived at almost the same time, their origins were vastly different. C, with its fraternal twin UNIX (TM), was born in one of the very bastions of computer science academia " Bell Labs " and its author, Dennis Ritchie, has ascended in the ensuing years to near-sainthood status. For a long time quality C compilers were only available in the UNIX environment, and while UNIX was fiendishly expensive to license for commercial users, AT practically gave it away to universities. Thus, an entire generation of computer science students and faculty were weaned and raised on UNIX and C and made it their true faith.

In contrast, Forth is literally a grassroots language; it has made its way in the world strictly on its own merits, lacking endorsements from the learned professors or distinguished looking textbooks from Addison-Wesley or

Springer-Verlag. Forth was invented by Charles Moore, a freelance programmer who had worked on control and data acquisition applications in environments ranging from a carpet factory to the Stanford Linear Accelerator Center. Moore was familiar with many programming languages, including assembly, Fortran, COBOL, PL/1, and APL, but felt that none of them met his needs well. He experimented extensively with interpreters in the early 1960s, eventually hitting on the idea of using a stack to pass parameters and results between the interpreted commands.

Moore recounts that his first interpreter which was recognizably Forth was written in Fortran on an IBM 1130 while he was working at a carpet manufacturer (Mohasco Industries) in 1968. He subsequently cross-compiled it to a Borroughs 5500, which was a CPU with push-down stacks that supported Forth much more gracefully, and then to a Univac 1108. Shortly afterward, Moore went to work for the National Radio Astronomy Observatory (NRAO) and ported Forth to a Honeywell 316. The language had been evolving rapidly during these ports and a compiler was added in the Honeywell version, resulting in the first Forth system in the modern sense and making it possible (eventually) to rewrite Forth in itself.

NRAO didn't appreciate the implications of Forth and, after some preliminary investigation into its patentability, released all rights to the language to Moore. Astronomers at Kitt Peak Observatory were soon using Forth on several Varian minicomputers for telescope control and data acquisition. In 1973, Moore, Elizabeth Rather (the second Forth programmer), and several others formed FORTH Inc. to sell Forth as a commercial product, originally called miniFORTH and later polyFORTH (TM). The language was soon running on several additional processors; the most important of these was the DEC PDP-11, which remains a mainstay of the FORTH Inc. user base today.

For the next five years, Forth the language and Forth the proprietary product of FORTH Inc. were one and the same. There were no other commercial vendors of Forth systems, and in fact the name Forth itself was trademarked. Although FORTH Inc. moved polyFORTH onto a broad variety of microprocessors as they became available, it maintained prices at mainframe levels. Except for a few scattered systems (descended from Moore's original NRAO implementation) in use at university astronomy departments, Forth was essentially inaccessable to casual or tentative users.In 1978, a small band of Forth enthusiasts formed the Forth Interest Group (FIG) in San Carlos, California. They assembled a team which, over a nine month period, designed a simple, transportable implementation of the Forth language, and then implemented it on the Intel 8080, DEC PDP-11, TI 9900, 6502, PACE, and Motorola 6800. This implementation, called FIG-Forth, was released into the public domain; Xeroxed listings were sold for a nominal fee ($10) and were disseminated worldwide in a matter of months. Some of the purchasers of the original listings performed further ports to the Intel 8086, the Nova, the RCA 1802, the Motorola 68000 and 6809, the Alpha Micro, the VAX, and even the Data General Eclipse, which were also placed in the public domain.

Although the FIG-Forth implementations were slow and crude by todays standards, they were perfectly suited to the times. In those days, RAM was dear, 5.25" floppy disks held 60-70 KB, and cassette tape or even paper tape was still a common mass storage device. A high-level interactive language with minimal memory requirements, available with complete source code, was the answer to many microcomputer experimenters' prayers. The FIG-Forth movement was given an additional impetus by the August 1980 issue of BYTE, which was devoted almost entirely to Forth articles and exposed Forth concepts to the general microcomputing community for the first time. The FIG-Forth listings also provided a launching pad for dozens of would-be Forth entrepeneurs. Since the listings reduced the ante for becoming a programming language vendor practically to zero, Forth software houses sprang up by the dozens, several for each microcomputer that was popular at the time. Although most of these Forth-vendor-wanna-bes did not survive, the influx of new blood brought about rapid improvements in Forth development tools. For example, prior to 1979, stand-alone Forth systems with line editors and integer arithmetic only were the rule. By 1981, Forth systems that ran as well-behaved tasks under a host operating system, used normal files for program and data storage, and supported floating point arithmetic and visual editors were readily available from the software houses which had gotten their start with FIG-Forth.

With the success of the FIG-Forth implementations under their belts, the founders of the Forth Interest Group looked for new fish to fry. The Forth Standards Team, a self-appointed spin-off from FIG, commenced deliberations to "improve" the language and issued the Forth-79 Standard in October 1980. Forth-79 was incompatible with both FIG-Forth and polyFORTH, failed to supplant either, and consequently had a most regrettable effect on Forth source code portability and the publishing of Forth textbooks and articles. Although it may seem difficult to believe, the Forth Standards Team proceeded to splinter the Forth community even further with the release of the Forth-83 Standard, which was (again) incompatible with all existing Forth dialects including Forth-79.

Forth-83, with all its problems, did have two highly desirable side-effects. First, the public domain FIG-Forth listings, which by 1983 were severely dated, were replaced virtually overnight by a popular public domain implementation of Forth-83 called F83, written by Henry Laxen and Michael Perry. F83 could recompile itself (FIG-Forth required an external assembler), had multitasking capabilities, ran under a host operating system, and generally represented a quantum jump in power and sophistication over FIG-Forth. (F83 was eventually superceded by a public domain multi-segmented Forth written by Tom Zimmer.) Second, rumblings about reactivation of the Forth Standards Team in 1987 caused major Forth vendors and users to band together and launch an ANSI standardization effort.

In the last few years, the Forth language vendors have undergone a severe shake-out, and only a half-dozen remain with solid product lines and significant user bases. (Judging by the number of companies who are managing to make a living in the much larger C marketplace, this may still be too many for the long haul.) FORTH Inc., with Elizabeth Rather as president, is clearly the flagship of the vendors; its revenues and staff exceed those of all the other Forth software houses taken together. Only FORTH Inc. has achieved the critical mass to offer a complete range of services: consulting, custom applications programming, education, and tools; the other vendors appear to subsist almost entirely on sales of packaged development systems.

While the pace of change has slowed in Forth programming tools, some exciting things have been happening on the hardware front. Several CPUs have been designed and built with Forth instruction sets, including Charles Moore's Novix NC 4016, Alan Winfield's MF1600, and Phil Koopman's WISC (Writable Instruction Set Computer) CPU/32. The performance of these machines is quite astonishing. For example, the Novix processor can crank out well over 4 MIPS, yet requires so little power and so few support chips that it can easily be built into a handheld computer.

Some Forth Terminology

Depending on who you talk to " or believe " Forth is a language, an environment, an operating system, and a philosophy of programming. All these claims are true, in some sense, but this semantic overload of the term Forth sometimes gives "outsiders" the sensation of being smothered in cotton wool when they try to understand what Forth is about. It doesn't help matters that Forth programmers use a unique " and somewhat bizarre " terminology when discussing how the language works and is structured.

Every Forth language element that can be used interactively or within a program is called a *word*. (The term *word* is also severely overloaded in Forth literature; in some contexts it can mean merely "a blank-delimited token from the input stream," and there is also a parsing operator named WORD). The source code for a particular word is called its *definition*. Some words are *primitives*, coded in the CPU's native machine language, the remainder are *high level or secondary definitions*, coded in Forth itself.

The Forth interpreter/compiler looks words up in the *dictionary*, which is essentially a symbol table that associates each word's name with the address of the corresponding executable code. The dictionary can be subdivided into *vocabularies* that can be searched independently or in a tiered fashion; this allows names to be overloaded and also allows "dangerous" or infrequently used words to be "hidden" so that they are not invoked accidentally.

Why do we concern ourselves with this strange nomenclature at all? For the purely pragmatic reason that to communicate with a Forth programmer, you need to use his terminology, because he probably won't understand yours. Terms like *word*, *dictionary*, *vocabulary*, and *definition* have the force of "tradition" behind them and pervade every Forth book, article, and set of source code in existence. Furthermore, many Forth programmers are self-taught, learned Forth only as a means to accomplish something else, and have never used any other programming language. Backgrounds in electrical or mechanical engineering, medicine, natural languages, music, astronomy, and the like are common; formal educations in computer science or mathematics are rare.

The Forth Language

Viewed strictly as a language, Forth is clearly in the camp of the modern, structured languages such as C, Pascal, and Modula II. It has all the proper flow of control constructs and, unlike C, does *no*t have a GOTO. Like C, but unlike Pascal and Modula II, Forth draws no distinction between procedures and functions. A more important difference between Forth and the other structured languages is that Forth has no support for data typing either at compile time or at runtime

.The key difference between Forth and the other structured languages is that the Forth compiler is *extensible*. Forth allows the programmer to declare new data types *and* allows him to define new compiler keywords (such as control structures) and then use them immediately " even within the very next procedure. Forth even has a compiler facility for creating new kinds of language *objects*, if you will: the programmer can separately define what happens at compile time (when the object is instantiated) and what happens at runtime (when the object is invoked). The closest C comes to being extensible is to allow you to build record formats out of existing data types with *struct*.

But Forth is not simply a set of keywords and syntax rules, it is a description of a virtual machine. The most important aspect of the virtual machine is that it has exactly two stacks: a parameter stack and a return stack. Nearly all Forth words use the parameter stack for both their arguments and their results; the return stack is used for flow of control within a program and (occasionally) for temporary storage of working values. Forth lends itself naturally to recursive algorithms because of this stack-based architecture; in "classic" Forth programming, global variables are frowned on and rarely used

.The extant Forth standards specify the names and actions of less than 200 words, which suffice to define the Forth virtual machine and the behavior of the interpreter/compiler. Commercial Forth development systems are much more elaborate, and have at least 500 words resident and available in the interpreter/compiler. Unfortunately, in many Forth manuals the words are simply documented in ASCII collating sequence, and are discussed as functional groups only as an afterthought (if at all). This makes the prospect of mastering a Forth system quite forbidding.

Actually, 500-600 language elements is almost exactly the same as the combined number of operators, keywords, and runtime library functions in Microsoft C or Borland C++, and you can study Forth *initially* with much the same approach as you would use for C. As a first approximation, you can view the relatively few "magic" words known to the Forth compiler as the "language," and treat the rest as a runtime library divided into functional categories such as stack operators, arithmetic/logical operators, memory access operators, and so on. As with any language, you will use 10% of the library 90% of the time, so you can just learn the most common functions first, and look up in the rest in the manual when you need them.

Once you've got a basic grasp of what's available in the Forth language, it's better to think about the system using a "layered" model, because this is the way the system is actually built up. The lowest layer consists of the primitives, which are written in the assembly language of the host CPU and implement the Forth virtual machine. The words coded as primitives tend to be the most frequently used arithmetic/logical, comparison, stack, and memory operators along with a few text-parsing building blocks for the interpreter/compiler. Each additional layer contains increasingly complex functions built out of the words defined in the layers below: console I/O, mass storage, formatting and string management, the interpreter, the compiler, and at the top, utilities such as the editor and assembler.

Two aspects of Forth that are frequently criticized are its cryptic names and its postfix (*or reverse Polish*) syntax. To address the former: when the fundamental Forth names were assigned, 10 character-per-second Teletypes were common and so were minicomputer systems with 8 or 16 KB of RAM. Consequently, short names were highly desirable to conserve memory and keystrokes, and we ended up with symbols such as @ for a memory fetch, ! for a memory store, and so on. I won't make any attempt to defend the historical Forth namings; each language has its conventions, and Forth's are no stranger than some found in C, LISP, or APL. Readable, maintainable programs can be written in any language, as can write-only, unmaintainable programs. The secret of obtaining the former rather than the latter is good design, discipline, and documentation, not use of a particular language. At least in Forth, since it is extensible, you can rename any language element to anything you like!

The complaints about about Forth's postfix syntax are more to the point, and deserve a more cogent response. In postfix systems, the arguments precede the operator; for example, to add 1 and 2 in Forth you would write:

```
1 2 +
```

whereas in C, which uses infix syntax, you would code:

```
1 + 2
```

and in LISP, which employs prefix syntax:

```
+ 1 2
```

When recursive descent compilers for C and Pascal parse your pretty infix expressions, they transform the expressions into an intermediate postfix form before generating machine code. Forth is postfix to start with to shift the burden of expression evaluation onto the programmer, so that the compiler can be simpler, smaller, and faster. While postfix notation certainly takes some getting used to, the success of HP calculators indicates that postfix should not be considered a major barrier to learning Forth. It is interesting to note that (because the Forth compiler is extensible) an infix expression evaluator can easily be layered onto the compiler; several have been published, but Forth programmers have apparently not found them necessary or useful.ul.

The Forth Environment

The C, Pascal, and Modula II languages are typically implemented as native code compilers, and true interpreters for these languages (when they exist at all) are usually slow and cumbersome. Although the so-called integrated environments for these languages, beginning with Turbo Pascal in 1983, have improved programmer productivity immensely, the environments are really just coresident editors, compilers, and linkers with a shell that allows the programmer to jump quickly from one to the other. They are not interpretive environments: you cannot invoke any language element, or a procedure you have written yourself, interactively by typing its name together with some parameters and look at the results.

Forth, on the other hand, is the original interpretive, interactive, integrated environment. In typical systems, the editor, assembler, and various debugging utilities are written in Forth itself and are kept resident at all times. The source code for these tools is usually kept on-line, so that the programmer can tune them to his personal taste.Although Forth has been implemented in many ways, the "classic" implementation is an unified interpreter and incremental compiler based on some startlingly simple concepts and ground rules. The core of the interpreter/compiler is a loop which reads blank-delimited tokens from the input stream (which can be either the keyboard or blocks of text from mass storage); the disposition of each token is controlled by the global variable STATE.

When STATE is FALSE, the system is said to be "interpreting." Each token is looked up in the dictionary; if found, the corresponding code is executed. If the token is not found, the interpreter attempts to convert it into a number which is left on the parameter stack; if the conversion is unsuccessful the interpreter issues an error message, clears the stack, and waits for a new command

.When STATE is TRUE, the system is "compiling" a new definition. If the token is found in the dictionary, some representation of the word (usually just a pointer) is compiled for later execution. If the token is not found but can be converted to a number, code is generated which will push that value onto the parameter stack when the new definition is executed. In other words, each "compiled" Forth definition is just a list of addresses and literal data, called "threaded code." The addresses refer, directly or indirectly, to executable machine code. When threaded code is "executed," a short routine called the "inner interpreter" walks down the lists of addresses and runs the corresponding machine language procedures.

As soon as a definition has been compiled, it can be interpreted " executed interactively by typing its name " or it can be referenced in subsequent definitions. All definitions in a Forth system have equal status, and there is no distinction or artificial barrier between the definitions that constitute the lower layers of the system (including the interpreter/compiler) and those added later as part of an application. An entire Forth program is run by just entering the name of its "highest definition."

There are two special classes of words which make the Forth compiler work: *defining words* and *immediate words*. Defining words create a new entry in the dictionary " that is, add a new symbol to the symbol table " and trigger the construction of a new variable, constant, procedure, or other object. The defining word that begins a new procedure " the colon (:) " changes STATE from false to true and thus initiates compilation. The application programmer creates new classes of Forth language objects by defining new defining words, and creates new members of the classes by invoking the new defining words.

Immediate words are so called because they are always executed immediately regardless of the value of STATE. The most important immediate words are the control structure words and the semicolon (;). The control structure words (such as IF or BEGIN) perform syntax checking and then generate appropriate code; for example, IF generates code which tests the value on top of the parameter stack and then performs a conditional branch on the result. The semicolon (;) is the terminator for a definition previously begun with the colon (:), its main action is to change the value of STATE back to FALSE. The application programmer extends the Forth compiler by defining new immediate words.

The Forth Operating System

For the first ten years of Forth's existence, it was always implemented and sold as a "native" system. That is, not only was it a programming environment, it was the whole environment. Native Forth systems contain their own device drivers and do not run under the control of a host operating system; they take over the entire machine. Moore's goals were speed and compactness at any price, and he was perfectly willing to give up the benefits of hardware independence and file systems in return.Forth interpreter/compilers that ran under host operating systems did not appear until 1979, in the first wave of small software houses that appeared at the time of the FIG-Forth listings. Two of the pioneers in this area were Laboratory Microsystems Inc., which sold a Z-80 Forth for CP/M that used operating system calls for I/O and file management, and Micromotion, which created a similar system for the Apple II. Over time, the advantages of host OS-based Forth interpreter/compilers became obvious, and today about 80% of the Forth programming systems sold fall into this category. So far as I am aware, FORTH Inc. is the only company still selling native Forth systems as a packaged product, but it now also produces systems that run under MS-DOS, RSX-11, and VAX VMS (among others).

In embedded applications, of course, Forth continues to be used in its native incarnation to reduce hardware requirements to a minimum. A special type of Forth compiler, called a target compiler, runs on a full-fledged development system and creates a stand-alone executable image (which can be ROMable). The image is then transferred to the target system by any convenient means " ROM, EPROM, downloading over a serial link, or diskette. Naturally, a target compiler can also be a cross compiler; since the target compiler itself is written in Forth, the CPU upon which the executable image is compiled need not be the same as the CPU upon which the image will be executed.

The Forth Philosophy

If Forth has a credo, it might be summed up as smallness, simplicity, and speed. Forth programmers traditionally disdain the flashy user interfaces and elaborate error handling that characterize most integrated environments nowadays. This attitude has its benefits " for example, a self-contained interactive Forth system including an editor, assembler, and even multitasking support can easily be put in an 8 KB EPROM. It also has its drawbacks " runtime error checking in a Forth system is virtually absent (unless it happens to come "for free," such as divide-by-zero detection on the Intel 80x86 family). Frequent total system crashes during Forth program development are taken for granted; indeed, they are expected.

Corollaries to the Forth credo are that intimate access to the hardware should always be available and portability should never get in the way of exploiting a CPU's instruction set to its fullest. A Forth system without an assembler is felt by most to be only slightly better than no Forth system at all. As for portability, you would be amazed at the number of arguments in the ANSI Forth Technical Committee that have hinged on whether a particular language feature can be implemented efficiently on someone's favorite processor.

Classic Forth programming style is characterized by many, many short, relatively simple definitions, bottom-up design, prototyping, and successive approximations. Forth dogma has it that an ideal definition should not exceed 2 or 3 lines and should not contain more than 9 or 10 elements. This ensures that individual definitions can be exhaustively tested; the programmer can simply put parameters on the stack and exercise the definition interactively through all of its possible internal paths. It also constrains an individual definition from becoming too specific, so that it is more likely to be reusable in the application.

The bottom up design of a Forth application derives naturally from the types of problems to which the language is routinely applied. A typical Forth application involves intimate interaction with hardware to acquire data or control a machine; in many cases, the hardware itself is only poorly understood or is not yet stable. The Forth words which are most hardware dependent are thus usually coded first; when these definitions are proven and the hardware is well understood, the rest of the application is built up by adding layers of increasing sophistication and complexity.

The emphasis on prototyping and successive approximations has much the same grounds as the stress on bottom-up design. In most Forth applications, getting the hardware to do what you want puts you a long ways toward your goal. After prototyping some code that approximates the final form of the application, you can simply discard or modify the upper layers until you get the specific functionality and user interface that you need; the lower, more hardware dependent levels of the application rarely require any significant change.

An important advantage of Forth in application design is that the documented user interface can rely on Forth's own interpreter, and merely consist of the top layer of definitions. This offers the sophisticated user incredible flexibility, because not only can he use the documented application entry points, he can extend the application by putting building blocks together in new ways, or access lower level words to test individual hardware components. In environments with naive users, the Forth compiler and the interpretation of "dangerous" Forth words are disabled, or an application-specific, self-contained user interface is written that does not lean on the Forth interpreter.

Forth's Place in the Universe

In 1969 there was exactly one Forth programmer: Charles Moore. When FORTH Inc. was founded in 1973, there were still probably less than a dozen. In 1979, Moore estimated that there were about 1000 Forth programmers. In 1988, based on the number of packaged systems that have been shipped by the larger vendors and the copies in print of Leo Brodie's Starting Forth (a popular Forth primer), I'd conjecture that there are as many as 50,000 serious Forth programmers and another 25,000-50,000 casual users. This is certainly respectable growth for a grassroots language. On the other hand, just to keep things in their proper perspective, something like 1.5 million copies of Turbo Pascal have been sold to date, never mind C compilers..

Looking back over Forth's somewhat checkered career, I can identify four distinct phases. I think of the first, spanning the years 1969 to 1978, as the "secret weapon" era. During this time, Forth evolved rapidly as it was used by a small set of people on highly specialized, demanding projects. There was no significant user base whose code would become obsolete when the language was changed, so many new concepts could be tried out and added to the language whenever they proved generally useful.

The second era, 1978 through 1982, marked the transition of Forth from a little-known proprietary product to a general purpose programming language. We could call this the "evangelical era" since it was marked by an intense proselytizing effort by FIG, augmented by Forth's position as the first inexpensive structured language with graphics, floating point, and its own editor and assembler on the immensely popular IBM PC (and later on the Macintosh). In retrospect, we owe the FIG founders a tremendous debt for legitimizing the language by seeding the "second-sources." Fortunately for us all, FORTH Inc. had the foresight not to enforce its trademark on the name "Forth;" I trust they have benefited more than they have suffered by this policy.

The years 1983 through 1986 were a period of Forth retrenchment and disillusionment. Most of the cultists, hobbyists, and nut cases drifted away from Forth as its novelty faded, and many of the professionals who continued to use Forth felt frustrated at its lack of penetration of the computer mainstream. They were convinced that they had a better solution for many classes of programming problems, but most computer science departments and the larger trade magazines continued to ignore Forth's existence. Smaller Forth vendors were driven out of business by advertising costs, the effort required to keep up with rapidly changing PC hardware, public domain Forths which could be downloaded from bulletin board systems, and the vendors' tendency to compete among themselves instead of trying to broaden the market.

In 1987, we entered the era of Forth maturation and true standardization. The ANSI Forth Technical Committee (X3J14), which operated under the auspices of both CBEMA and the IEEE Computer Society, made slow but steady progress reconciling the existing Forth dialects, and the after many revisions the committee's work was finally approved as the American National Standard for Forth Programming Language in the Spring of 1994. While an ANSI Forth Standard will not be an instant cure for all Forth's ills, it should drastically improve portability of both Forth applications and Forth programmers, and thus make Forth a more viable candidate for consideration by project planners.

Forth development systems are available for nearly every CPU in existence, and Forth is consistently the first third-party high level language to appear on a new CPU architecture, personal computer, or operating system. Forth continues to enjoy wide use in real time control and data acquisition applications, and can be found in the most unexpected places: from the bar-code-reader "wand" carried by your friendly Federal Express driver, to the American Airlines baggage control system, to instrumentation for SpaceLab experiments, to a networked monitoring and control system with some 400 processors and 17,000 sensors at an airport in Saudi Arabia. While I think it likely that Forth will always remain a somewhat obscure language, its speed and interactivity coupled with minimal hardware requirements and ready portability are a unique combination of assets. In its niche, Forth has no real competition, and I expect it will be around for a long time to come.

This section was originally published as "A Forth Apologia" in Programmer's Journal, Volume 6, Number 6, November/December 1988, page 56.

back to the software page

back to the home page