XIII

Forth Session

Chair: *Helen Gigley*

THE EVOLUTION OF FORTH

Elizabeth D. Rather
FORTH, Inc.
111 N. Sepulveda Blvd., #300
Manhattan Beach, CA 90266

Donald R. Colburn
Creative Solutions, Inc.
4701 Randolph Rd., Suite #12
Rockville, MD 20853

Charles H. Moore
Computer Cowboys
410 Star Hill Road
Woodside, CA 94062

ABSTRACT

Forth is unique among programming languages in that its development and proliferation has been a grass-roots effort unsupported by any major corporate or academic sponsors. Originally conceived and developed by a single individual, its later development has progressed under two significant influences: professional programmers who developed tools to solve application problems and then commercialized them, and the interests of hobbyists concerned with free distribution of Forth. These influences have produced a language markedly different from traditional programming languages.

CONTENTS

13.1 CHUCK MOORE'S PROGRAMMING LANGUAGE

Forth was invented by Charles H. (Chuck) Moore. A direct outgrowth of Moore's work in the 1960s, the first program to be called Forth was written in about 1970 [Moore 1970a]. This section covers the early work leading to Forth.

13.1.1 Early Development

Moore's programming career began in the late 1950s at the Smithsonian Astrophysical Observatory with programs to compute ephemerides, orbital elements, satellite station positions, and so forth

[Moore 1958; Veis, 1960]. His source code filled two card trays. To minimize recompiling this large program, he developed a simple interpreter to read cards controlling the program. This enabled him to compose different equations for several satellites without recompiling. This interpreter featured several commands and concepts that survived into modern Forth, principally a command to read "words" separated by spaces, and one to convert numbers from external to internal form, plus an **IF ... ELSE** construct. He found free-form input to be both more efficient (smaller and faster code) and reliable than the more common FORTRAN practice of formatting into specific columns, which had resulted in numerous reruns due to misaligned columns.

In 1961, Moore received his BA in Physics from MIT and entered graduate school at Stanford. He also took a part-time programming position at the Stanford Linear Accelerator (SLAC), writing code to optimize beam steering for the (then) pending two-mile electron accelerator, using an extension of some of his prior work with least-squares fitting. A key outgrowth of this work was a program called **CURVE**, coded in ALGOL (1964), a general-purpose nonlinear differential-corrections data fitting program. To control this program, he used an enhanced version of his interpreter, extended to manage a push-down stack for parameter passing, variables (with the ability to explicitly fetch and store values), arithmetic and comparison operators, and the ability to define and interpret procedures.

In 1965, he moved to New York City to become a free lance programmer. Working in FORTRAN, ALGOL, JOVIAL, PL/I and various assemblers, he continued to use his interpreter as much as possible, literally carrying around his card deck and recoding it as necessary. Minicomputers appeared in the late '60s, and with them teletype terminals, for which Moore added operators to manage character input and output. One project involved writing a FORTRAN-ALGOL translator and file-editing utilities. This reinforced for him the value of spaces between words, which were not required in FORTRAN source.

Newly married and seeking a small town environment, Moore joined Mohasco Industries in Amsterdam, NY, in 1968. Here he developed computer graphics programs for an IBM 1130 minicomputer with a 2250 graphic display. This computer had a 16-bit CPU, 8K RAM, his first disk, keyboard, printer, card reader/punch (used as disk backup!), and FORTRAN compiler. He added a cross-assembler to his program to generate code for the 2250, as well as a primitive editor and source-management tools. This system could draw animated 3-D images, at a time when IBM's software for that configuration drew only static 2-D images. For fun, he also wrote a version of *Spacewar*, an early video game, and converted his ALGOL Chess program into the new language, now (for the first time) called FORTH. He was impressed by how much simpler it became.

The name FORTH was intended to suggest software for the fourth (next) generation computers, which Moore saw as being characterized by distributed small computers. The operating system he used at the time restricted file names to five characters, so the "U" was discarded. FORTH was spelled in upper case until the late '70s because of the prevalence of upper-case-only I/O devices. The usage "Forth" was generally adopted when lower case became widely available, because the word was not an acronym.

Moore found the Forth-based 1130 environment for programming the 2250 superior to the FORTRAN environment in which the 1130 software was developed, so he extended it into an 1130 compiler. This added looping commands, the concept of keeping source in 1024-byte blocks and tools for managing them, and most of the compiler features we recognize in Forth today.

Most important, there was now a dictionary. Procedures now had names, and the interpreter searched a linked list of names for a match. Names were compiled with a count and three characters, a practice learned from the compiler writers of Stanford and which prevailed in Forth until the 1980s. Within a dictionary entry was a "code field" containing the address of code to be executed for that

routine. This was an indirect threaded code implementation (see Section 13.5.2) and was in use five years before Dewar's paper on indirect threaded code appeared in *Communications of the ACM* [Dewar 1975]. The use of indirect threaded code was an important innovation, because an indirect jump was the only overhead once a word had been found. Dictionary entries could consist either of pointers to other "high-level" routines or of machine instructions.

Finally, in order to provide a simple mechanism for nesting routines, a second stack called the "return stack" was added. The benefit of having a stack reserved for return addresses was that the other stack could be used freely for parameter passing, without having to be "balanced" before and after calls.

The first paper on Forth was written at Mohasco [Moore 1970a].

In 1970, Mohasco assigned Moore to an ambitious project involving a new Univac 1108, handling a network of leased lines for an order-entry system. He ported Forth onto the 1108, and arranged for it to interface to COBOL modules that did the transaction processing. The 1108 Forth was coded in assembler. It buffered input and output messages and shared the CPU among tasks handling each line. It also interpreted the input and executed the appropriate COBOL modules. This version of Forth added mechanisms for defining and managing tasks, and also added an efficient scheme for managing disk block buffers similar to schemes in use today.

Unfortunately, an economic downturn led Mohasco to cancel the 1108 project before completion. Moore immediately gave notice, then wrote an angry poem and a book on Forth [Moore 1970b] that was never published. It described how to develop Forth software and encouraged simplicity and innovation.

13.1.2 Philosophy and Goals

To Moore, Forth was a personal response to his frustration with existing software tools, which he viewed as a sort of "Tower of Babel":

> The software provided with large computers supplies a hierarchy of languages: the assembler defines the language for describing the compiler and supervisor; the supervisor the language for job control; the compiler the language for application programs; the application program the language for its input. The user may not know, or know of, all these languages: but they are there. They stand between him and his computer, imposing their restrictions on what he can do and what it will cost.
>
> And cost it does, for this vast hierarchy of languages requires a huge investment of man and machine time to produce, and an equally large effort to maintain. The cost of documenting these programs and of reading the documentation is enormous. And after all this effort the programs are still full of bugs, awkward to use and satisfying to no one. [Moore 1970a]

Moore conceived of Forth as replacing the entire "vast hierarchy" with a single layer, requiring only two elements: a programmer-to-Forth interface, consisting of minimal documentation (minimal because the interface should be simple and natural), and the Forth-machine interface, consisting of the program itself.

His view was entirely personal, considering his own needs in light of his own experience. The following excerpts from his unpublished book [Moore 1970b], describe this view:

> I've written many programs over the years. I've tried to write *good* programs, and I've observed the manner in which I write them rather critically. My goal has been to decrease the effort required and increase the quality produced.

> In the course of these observations, I've found myself making the same mistakes repeatedly. Mistakes that are obvious in retrospect, but difficult to recognize in context. I thought that if I wrote a prescription for programming, I could at least remind myself of problems. And if the result is of value to me, it should be of value to others....

Above all, his guiding principle, which he called the "Basic Principle," was, "Keep it simple!" Throughout his career he has observed this principle with religious dedication.

> As the number of capabilities you add to a program increases, the complexity of the program increases exponentially. The problem of maintaining compatibility among these capabilities, to say nothing of some sort of internal consistency in the program, can easily get out of hand. You can avoid this if you apply the Basic Principle. You may be acquainted with an operating system that ignored the Basic Principle.
>
> It is very hard to apply. All the pressures, internal and external, conspire to add features to your program. After all, it only takes a half-dozen instructions, so why not? The only opposing pressure is the Basic Principle, and if you ignore it, there is no opposing pressure.

The main enemy of simplicity was, in his view, the siren call of generality that led programmers to attempt to speculate on future needs and provide for them. So he added a corollary to the Basic Principle: "Do not speculate!"

> Do not put code in your program that *might* be used. Do not leave hooks on which you can hang extensions. The things you might want to do are infinite; that means that each has 0 probability of realization. If you need an extension later, you can code it later—and probably do a better job than if you did it now. And if someone else adds the extension, will he notice the hooks you left? Will you document this aspect of your program?

This approach flew in the face of accepted practice then, as now. A second corollary was even more heretical: "Do it yourself!"

> The conventional approach, enforced to a greater or lesser extent, is that you shall use a standard subroutine. I say that you should write your own subroutines.
>
> Before you can write your own subroutines, you have to know how. This means, to be practical, that you have written it before; which makes it difficult to get started. But give it a try. After writing the same subroutine a dozen times on as many computers and languages, you'll be pretty good at it.

Moore followed this to an astounding extent. Throughout the '70s, as he implemented Forth on eighteen different CPUs (Table 13.1). He invariably wrote for each his own assembler, his own disk and terminal drivers, even his own multiply and divide subroutines (on machines that required them, as many did). When there were manufacturer-supplied routines for these functions, he read them for ideas, but never used them verbatim. By knowing exactly how Forth would use these resources, by omitting hooks and generalities, and by sheer skill and experience (he speculated that most multiply/divide subroutines were written by someone who had never done one before and never would again), his versions were invariably smaller and faster, usually significantly so.

Moreover, he was never satisfied with his own solutions to problems. Revisiting a computer, or an application, after a few years, he often rewrote key code routines. He never reused his own code without re-examining it for possible improvements. This later became a source of frustration to Rather, who, as the marketing arm of FORTH, Inc. (see Section 13.2.2), often bid jobs on the assumption that inasmuch as Moore had just done a similar project, this one would be easy—only to watch helplessly as he tore up all his past code and started over.

Today, Moore is designing Forth-based microprocessors using his own Forth-based CAD system, which he has rewritten (and sometimes rebuilt, with his own hardware) almost continuously since 1979.

TABLE 13.1

Table showing computers for which Chuck Moore personally implemented Forth systems. In 1978, his implementations of Forth on the Level 6 and 8086 represented the first resident software on both CPUs, anticipating their manufacturers' systems by many months.

Year	Model	Customer	Forth Applications
1970–71	Honeywell H316	National Radio Astronomy Observatory (NRAO)	Data acquisition, on-line analysis w/ graphics terminal
1971	Honeywell DDP116	NRAO	Radio telescope control
1971–2	IBM 370/30	NRAO	Data analysis
1972	Varian 620	Kitt Peak National Observatory (KPNO)	Optical telescope control and instrumentation
1972	HP2100	KPNO	Instrumentation
1972–3	Modcomp	NRAO	Data analysis
1973	PDP-11	NRAO	Radio telescope control, data acquisition, analysis, graphics
1973	DG Nova	Steward Observatory	Data acquisition and analysis
1974	SPC-16	Steward Observatory	Ground control of balloon-borne telescope
1975	SDS920	Aerospace Corp.	Antenna control
1975	Prime	Gen'l Dynamics, Pomona	Environmental controls
1976	Four-Phase	Source Data Systems	Data entry and database management
1977	Interdata Series 32	County of Alameda, CA	Database management
1977	CA LSI-4	MICOA	Business systems
1978	Honeywell Level 6	Source Data Systems	Data entry and database management
1978	Intel 8086	Aydin Controls	Graphics and Image Processing
1980	Raytheon PTS-100	American Airlines	Airline display and workstations

Moore considered himself primarily an applications programmer, and regarded this as a high calling. He perceived that "systems programmers" who built tools for "applications programmers" to use had a patronizing attitude toward their constituents. He felt that he had spent a great portion of his professional life trying to work around barriers erected by systems programmers to protect the system from programmers and programmers from themselves, and he resolved that Forth would be different. Forth was designed for a programmer who was intelligent, highly skilled, and professional; it was intended to empower, not constrain.

The net result of Moore's philosophy was a system that was small, simple, clean—and extremely flexible: in order to put this philosophy into practice, flexible software is essential. The reason people leave hooks for future extensions is that it's generally too difficult and time-consuming to re-implement something when requirements change. Moore saw a clear distinction between being able to teach a computer to do "anything" (using simple, flexible tools) and attempting to enable it to do "everything" with a huge, general-purpose OS. Committing himself to the former, he provided himself with the ideal toolset to follow his vision.

13.2 DEVELOPMENT AND DISSEMINATION

By the early 1970s, Forth had reached a level of maturity that not only enabled it to be used in significant applications, but that attracted the attention of other programmers and organizations. Responding to their needs, Moore implemented it on more computers and adapted it to handle ever larger classes of application.

13.2.1 Forth at NRAO

Moore developed the first complete, stand-alone implementation of Forth in 1971 for the 11-meter radio telescope operated by the National Radio Astronomy Observatory (NRAO) at Kitt Peak, Arizona. This system ran on two early minicomputers (a 16 KB DDP-116 and a 32 KB H316) joined by a serial link. Both a multiprogrammed system and a multiprocessor system (in that both computers shared responsibility for controlling the telescope and its scientific instruments), it was responsible for pointing and tracking the telescope, collecting data and recording it on magnetic tape, and supporting an interactive graphics terminal on which an astronomer could analyze previously recorded data. The multiprogrammed nature of the system allowed all these functions to be performed concurrently, without timing conflicts or other interference.

The system was also unique for that time in that, all software development took place on the minis themselves, using magnetic tape for source. Not only did these Forth systems support application development, they even supported themselves. Forth itself was written in Forth, using a "metacompiler" to generate a new system kernel when needed.

To place these software capabilities in context, it's important to realize that manufacturer-supplied system software for these early minicomputers was extremely primitive. The main tools were cross-assemblers and FORTRAN cross-compilers running on mainframes (although the FORTRAN cross-compilers were too inefficient to do anything complex, given the tiny memories on the target machines). On-line programming support was limited to assemblers loaded from paper tape, with source maintained on paper tape. Digital Equipment Corporation had just announced its RT-11 OS for its PDP-11 line, which offered limited foreground-background operation; no form of concurrency was available for the H316 family. Multi-user operation of the sort that enabled NRAO's astronomers to graphically analyze data while an operator controlled the telescope and live data was flowing in, was unheard of.

Edward K. Conklin, head of the Tucson division of NRAO, which operated the 11-meter telescope, found it difficult to maintain the software as Moore was based at NRAO's headquarters in Charlottesville, VA. So, in 1971, he brought in Elizabeth Rather, a systems analyst at the University of Arizona, to provide local support on a part-time basis. Rather was appalled to find this critical system written in a unique language, undocumented, and known to only one human. Her instinctive reaction was to rewrite the whole thing in FORTRAN to get it under control. Alas, however, there was neither time nor budget for this, so she set out to learn and document the system, as best she could.

After about two months, Rather began to realize that something extraordinary was happening: despite the incredibly primitive nature of the on-line computers, despite the weirdness of the language, despite the lack of any local experts or resources, she could accomplish more in the few hours she spent on the Forth computers once a week than in the entire rest of the week when she had virtually unlimited access to several large mainframes.

She wondered why. The obvious answer seemed to lie in the interactive nature of Forth. (The programmer's attention is never broken by the procedural overhead of opening and closing files, loading and running compilers, linkers, loaders, debuggers, and the like. But there's more to it than

that. For example, all the tools used by Forth's OS, compiler, and other internal functions are available to the programmer. And, as Chuck Moore intended, its constraints are minimal and its attitude is permissive. Forth devotees still love to debate the source and magnitude of such productivity increases!)

Rather immediately left the University and began working for NRAO jointly with Kitt Peak National Observatory (KPNO), an optical observatory with which NRAO shared facilities, maintaining the Forth system for NRAO and developing one for KPNO (which was later used on KPNO's 156" Mayall telescope and other instruments [Phys. Sci. 1975]). During the next two years she wrote the first Forth manual [Rather 1972] and gave a number of papers and colloquia within the observatory and related astronomical organizations [Moore 1974a].

In 1973, Moore and Rather replaced the twin-computer system by a single disk-based PDP-11 computer [Moore 1974a&b]. This was a multi-user system, supporting four terminals, in addition to the tasks of controlling the telescope and taking data. It was so successful that the control portions of it were still in use in 1991 (data acquisition and analysis functions are more dependent on experimental equipment and techniques, which have changed radically over the years). The system was so advanced that astronomers from all over the world began asking for copies of the software. Versions were installed at Steward Observatory, MIT, Imperial College (London), the Cerro Tololo (Chile) Inter-American Observatory, and the University of Utrecht (Netherlands). Its use spread rapidly, and in 1976 Forth was adopted as a standard language by the International Astronomical Union.

13.2.2 Commercial Minicomputer Systems

Following completion of the upgraded system in 1973, Moore and his colleagues Rather and Conklin formed FORTH, Inc. to explore commercial uses of the language. FORTH, Inc. developed multi-user versions of Forth [Rather 1976a] for most of the minicomputers then in use (see Table 13.1), selling these as components of custom applications in a widely diverse market, ranging from database applications to scientific applications, such as image processing. The minicomputers and applications of the '70s provided the environment in which Forth developed and stabilized, to the extent that all the innovations contributed by independent implementors in the years that followed, represented relatively minor variants on this theme. Because of this, we shall take a close look at the design and structure of these systems.

13.2.2.1 *Environmental constraints*

Minicomputers of the 1970s were much less powerful than the smallest microcomputers of today. In the first half of the decade not all systems even had disks—1/2" tape was often the only mass storage available. Memory sizes ranged from 16 to 64 Kbytes, although the latter were considered large. In the early '70s, most programming for minis was done in assembly language. By the middle of the decade, compilers for FORTRAN and BASIC were available, and manufacturer-supplied executives such as DEC's RT-11 supported foreground-background operation. Multi-user systems were also becoming common: a PDP-11 or Nova could be expected to support up to eight users, although the performance in a system with eight active users was poor.

On this hardware, Moore's Forth systems offered an integrated development toolkit including interactive access to an assembler, editor, and the high-level Forth language, combined with a multitasking, multi-user operating environment supporting 64 users without visible degradation, all resident without run-time overlays.

Although time-critical portions of the system were written in assembler, as most applications required very high performance, Moore could port an entire Forth development environment to a new computer in about two weeks. He achieved this by writing Forth in Forth—any Forth computer could generate Forth for another, given the target system's assembler and code for about 60 primitives. Because the first step in a port was designing and writing the target assembler, it is possible that Moore has written more assemblers for different processors than anyone else.

Being able to port the system easily to new architectures was important, as the minicomputer market was extremely fragmented. A large number of CPUs was available, and each was supported by a large number of possible disk controller and drive combinations. Today, by contrast, the microcomputer market is dominated by a very short list of processor families, and adherence to *de facto* standards such as the PC/AT is the norm.

Installations were done on site, because it was impractical to ship the minicomputers. When LSI-11s first became available, Moore bought one and mounted it in a carry-on suitcase, with a single 8" floppy drive in a second suitcase. This portable personal computer accompanied him everywhere until 1982, acting as a "friendly" host for generating new Forths.

13.2.2.2 *Application Requirements*

If the principal environmental constraints were memory limitations and a need to serve a broad spectrum of CPU architectures, the application requirements were dominated by a need for performance. Here are some of the principal application areas in which Forth achieved success in this period:

1. **Commercial/business data base systems:** First developed for Cybek Corporation under the guidance of Arthur A. Gravina, these systems supported multiple terminals on a Data General Nova, handling high-speed transaction processing. The first was written for Vernon Graphics, Inc., a service bureau to Pacific Telephone, in 1974. It supported 32 terminals processing transactions against a 300 MB database. In its first week the system handled over 100,000 transactions a day (40,000 was the requirement). The system was subsequently upgraded to support 64 terminals and a 600 MB database, with no discernable degradation in response times, which remained under one second.

   Cybek subsequently marketed this system for business applications in banking and hospital management; its current version is marketed by a division of McDonnell Douglas. A similar effort by Source Data Systems in Iowa produced a multi-terminal data-entry system marketed by NCR Corp. for hospital management and similar applications.

   The performance of such a system is overwhelmingly dominated by operating system issues, principally the ability of the native Forth block-based file system to read and write data files very quickly.

2. **Image Processing:** FORTH, Inc. developed a series of image processing applications for the Naval Weapons Research Center, NASA's Goddard Space Flight Center, the Royal Greenwich Observatory in England, and others. Central to these was a need for performing standardized operations (e.g., enhancement, windowing, etc.) on images residing on different kinds of hardware. The approach taken included many features now associated with object-oriented programming: encapsulation (the basic object was an "image," with characteristic parameters and methods), inheritance (you could add new images that would inherit characteristics of previously defined classes of images) and dynamic binding of manipulation methods. Moore, the principal architect of this approach, was unaware of any academic work in this area. Striving to achieve the same goals as later OOPS writers, he independently derived similar solutions.

Image processing systems are also distinguished by a need to manipulate and move large quantities of data very fast; a 512x512x16 image, for example, occupies 512 KB. In addition to the high-speed disk performance that characterized Forth database systems, these also required fast processing speed and the ability to handle algorithms such as FFTs. As many minicomputers lacked hardware floating-point arithmetic, Forth included flexible integer and fixed-point fraction operators, as well as specialized array primitives.

3. **Instrumentation and control:** Forth was first developed and used for this purpose at NRAO, and Forth is widely used for instrumentation and controls today. FORTH, Inc. produced several more astronomical systems (for the Universities of Wyoming [Gehrz 1978], Minnesota, Hawaii and Illinois; Cal Tech; plus the Royal Greenwich Observatory and St Andrews University in the UK). In addition, a number of commercial instrument manufacturers, such as Princeton Applied Research (now a division of EG&G) and Nicolet Instruments, adopted Forth as a language for internal development.

   These applications are characterized by high data rates, as much as 20 KHz in some cases, which really strained the CPU speed of the processors available. Fast interrupt response was essential, along with high-speed multitasking to allow data acquisition to proceed concurrently with operator activity and instrument control.

13.2.2.3 *Influences*

The evolution of Forth prior to 1978 was completely dominated by Moore himself. As we have seen, Moore was, and is, a fanatic minimalist, dedicated to the principle of zero-based design in which every feature, and every instruction, must justify its existence or be ruthlessly scrapped.

Moore originally developed the system for his own use. It surprised him a little to find that Rather, and the other early users, also liked it and found it enhanced their productivity as much as it did his. But even after the formation of FORTH, Inc. and its open marketing of the system, the selection and design of support tools and the general programming interface was dominated by his personal tastes.

Moore was working primarily as a consultant, supported by others within FORTH, Inc., installing a Forth system on a customer's computer as the first step in developing a custom application. Because the customer was primarily interested in the application, it was imperative that the port be completed quickly and inexpensively. The extreme simplicity of Forth made this possible without compromising the performance of the application.

Each of these projects contributed its own lessons, tools, and techniques. Moore carried microfiche listings of all previous projects in his briefcase, and often referred to them to get the code for some unique primitive or driver from the past. Frequently used words might become a standard fixture of the system. Also, improved techniques for solving common problems were integrated into the system.

This pattern of continual evolution created customer support headaches for FORTH, Inc., however, as no two installed systems were the same. In most cases the installation included a five-day Forth programming course taught by Rather, who had to check every evening to make sure that the system still behaved the way it was being taught.

13.2.3 Early Microprocessor Systems

In 1976, Robert O. Winder, of RCA's Semiconductor Division, engaged FORTH, Inc. to implement Forth on its new CDP-1802 8-bit microprocessor [Rather 1976b; Electronics 1976]. The new product, called "microFORTH," was subsequently implemented on the Intel 8080, Motorola 6800, and Zilog Z80, and sold by FORTH, Inc. as an off-the-shelf product. microFORTH was successfully used in

numerous embedded microprocessor instrumentation and control applications in the United States, Britain, and Japan.

13.2.3.1 *Environment and Applications*

microFORTH was FORTH, Inc.'s first experience with off-the-shelf, mail-order software packages; the minicomputer systems were all installed on-site. The mail-order operation was made possible by the rapid standardization of the industry on 8" "IBM-format" floppy disks, and the relatively small number of development systems for each CPU type.

These microprocessors were all 8-bit devices, typically with 16K bytes of memory in the development system. The target systems were usually custom boards (although Intel's Single Board Computer series quickly became popular), and the software was expected to run from PROM in an embedded environment without disk or (usually) terminal. This was significantly different from the minicomputer environment, where there was always a disk, and a program was expected to run on the same (or identical) computer as the one used for development.

Most microprocessor manufacturers offered development platforms consisting of the same microprocessor as in the target, up to 64K bytes of RAM, a serial line for a terminal, a parallel printer port, and two 8" floppy disk drives. Software support was mainly assembler, although Intel soon introduced PL/M. In-circuit emulators and separate utilities were introduced for debugging.

microFORTH was principally marketed as an interactive alternative to assembler which, unlike PL/M, was available across most microprocessor families and therefore offered a higher degree of transportability.

13.2.3.2 *Language Definition*

Following some initial experimentation with 8-bit stack width and 128-byte block buffers, it was quickly decided to maintain the same basic internal architecture as on the minicomputer systems. The organization of the program changed significantly, however.

microFORTH came with a target nucleus designed to run from PROM. This nucleus was only 1K in size, containing primitives such as single-precision arithmetic and other very basic functions. The development environment supported writing and testing code interactively, and then compiling a version of that code designed to mate to the run-time nucleus. A version of **VARIABLE** was provided to support segregated ROM/RAM data space (**CONSTANT**s were in PROM), and defining words were adapted so that user-defined structures could be made to reside in either. And whereas previously, **VARIABLE**s could be initialized at compile time, that capability was removed, as it is difficult to initialize target RAM when a ROM is being compiled without setting up a "shadow" table: ROM space was considered too precious for that.

The multiprogramming support was initially stripped out, although it later came back using a new, faster task-swapping algorithm, and the database tools vanished completely.

FORTH, Inc. never released the metacompiler used to generate Forth on new minicomputer CPUs. A variant of this metacompiler became an integral part of microFORTH, however, as it was used to generate the ROMable code for the target application. This was significant, as we shall see in the next section.

13.2.3.3 *Influences*

The principal architect of microFORTH was Dean Sanderson. Although Sanderson worked closely with Moore and shared most of his basic philosophies, differences in style and approach were

inevitable. But the major new influence came from the broader customer base that resulted from the wider marketing of microFORTH. It was customer pressure that brought back multiprogramming, and this larger customer base also caused standards groups to form.

13.2.4 Language Definition

The commercial mini and microcomputer implementations produced by FORTH, Inc. in the early- and mid-1970s, for the first time encapsulated the principles and elements of Forth as it is used today. For this reason, we shall summarize these briefly.

13.2.4.1 *Design Principles*

Much as algebra was the "metaphor" for FORTRAN, Forth was conceived on the model of English prose (though some have suggested that its postfix notation tends to resemble verb-at-the-end languages such as German). Its elements ("words") are named data items (roughly equivalent to nouns), named procedures (equivalent to verbs), and defining words (special kinds of verbs capable of creating data items with customized characteristics). Words may be defined in terms of previously defined words or in machine code (using the embedded assembler).

Forth "words" are functionally analogous to subroutines in other languages. They are also equivalent to commands in other languages—Forth blurs the distinction between linguistic elements and functional elements.

Words are referenced (either from the keyboard or in program source) by name. As a result, the term "word" is applied both to program (and linguistic) units and to their text names. In parsing text, Forth considers a word to be any string of characters bounded by spaces (or "white space" characters in some file-based systems). Except for these, there are no special characters that cannot be included in a word or start a word, although many programming teams adopt naming conventions to improve readability. Words encountered in text fall into three categories: defined words (i.e., Forth routines), numbers, and undefined words.

There are no explicit typing mechanisms in Forth, a feature that sometimes surprises newcomers, but is generally admired by experienced Forth programmers.

13.2.4.2 *Structured Programming Disciplines*

Architecturally, Forth words adhere strictly to the principles of "structured programming" as articulated by Dijkstra [Dijkstra 1970] and "modular programming" [Parnas 1972]. These principles may be summarized as follows:

- Every program is described as a linear sequence of self-contained modules;
- A module has one entry point, one exit point, and ideally performs one function, given a set of inputs and a set of outputs;
- A module can contain:
  - — references to other modules;
  - — decision structures (**IF THEN** statements);
  - — looping structures.

Top-down design and bottom-up coding and testing are strongly encouraged by Forth's structure.

As was the case with Moore's independent development of OOPs-like features in his image processing system, Moore was unfamiliar with the contemporary literature on structured programming. These principles were first called to his attention in 1973 by Rather, who received several comments on the apparent relationship between Forth and structured programming in seminars she was giving on Forth. On reading one of Djikstra's papers, Moore observed, "it just seems like good programming practice to me."

In fact, advanced Forth programmers with knowledge of the underlying implementation know ways of "cheating," but such practices are frowned upon, and definitely not supported or encouraged by the structure of the language.

13.2.4.3 *Elements of Forth*

Moore's Forth systems of the early 1970s were built on a nucleus of only 4K bytes. This tiny program included disk (or tape) and terminal drivers and the ability to search and build the dictionary. This nucleus was then used to compile from source the balance of the programming environment, including the assembler, editor, multiuser support, and several hundred general commands. Booting the system, including compiling most of it from source into executable form, took only a few seconds.

A metacompiler, also written in Forth, was used to compile the nucleus. The entire source for the system was about 40 pages long.

These systems were "native," that is, running without any host OS or executive. This was a necessity in the early days, as OSs weren't available. Later, it was regarded as a significant advantage, as I/O services in a native Forth environment were much faster than could be supplied by a general purpose OS.

The principal elements of Forth are discussed briefly in the sections that follow.

Dictionary: A Forth program is organized into an extensible dictionary that occupies almost all the memory used by the system. The dictionary is classically implemented as a linked list of variable-length items, each of which defines a word. The content of each definition depends upon the type of word (data item, constant, sequence of operations, etc.). On multi-user Forth systems, individual users may have private dictionaries, each of which is connected to a shared, re-entrant system dictionary.

Push-Down Stacks: Forth maintains two push-down stacks, or LIFO lists (on a multiprogrammed version, a pair for each task). These are used to pass data between Forth words and for controlling logical flow. A stack contains one-cell items, where a cell is 16 bits wide on 8-bit and 16-bit computers, and 32 bits wide on most implementations for 32-bit processors such as the 680x0 family. Extended-precision numbers occupy two stack positions, with the most significant part on top. Items on either stack may be addresses or data items of various kinds. Stacks are of indefinite size, and usually grow towards low memory.

Forth's explicit use of stacks leads to a "postfix" notation in which operands precede operators. Because results of operations are left on the stack, operations may be strung together effortlessly, and there is little need to define variables to use for temporary storage.

Interpreters: Forth is an interpretive system, in that program execution is typically controlled by a small machine-code routine (often only two or three instructions) interpreting lists of pointers or tokens for abstract machine functions. This architecture is much faster than classical interpreters, as used in BASIC and PROLOG for example, enabling it to perform satisfactorily in the real-time applications for which it was designed.

This internal engine is often referred to as the "inner" or "address" interpreter, as distinct from Forth's more traditional text interpreter which processes source and user input. The text interpreter extracts strings separated by spaces from the terminal or mass storage, looking each word up in the dictionary. If a word is found it is executed by invoking the address interpreter, which processes a string of addresses compiled in a word definition by executing the definition pointed to by each. The text is not stored in memory, even in condensed form. If a word is not found, the system attempts to convert it as a number and push it onto the stack. If number conversion fails (due to a nonnumeric character), the interpreter aborts with an error message.

The address interpreter has two important properties. First, it is fast, often requiring as few as one or two machine instructions per address. Second, it makes Forth definitions extremely compact, as each reference requires only one cell (or computer word; Forth users prefer to avoid the use of "word" as a hardware unit because of its use to denote an element in the language). In contrast, a subroutine call constructed by most compilers requires instructions for handling the calling sequence before and after a **CALL** or **JSR** instruction and address, and typically, **save** and **restore** registers within the subroutine. Forth's stack architecture obviates the need for an explicit calling sequence, and most implementations make global register assignments, in which certain system state variables are assigned to dedicated registers, and all other registers are designated scratch registers for use in code words.

Assembler: Most Forth systems include a macro assembler for the CPU on which they run. When using **CODE**, the programmer has full control over the CPU, as with any other assembler, and **CODE** definitions run at full machine speed. The assembler lets the programmer use explicit CPU-dependent code in manageable pieces with machine-independent interfacing conventions. To move an application to a different processor requires recoding only the **CODE** words, which will interact with other Forth words in exactly the same manner.

Forth assemblers feature an unusual design, which has two goals: (1) to improve transportability between processors by standardizing assembler notation as much as possible without impairing the programmer's control of the processor; and (2) to yield a compact assembler that can be resident at all times to facilitate interactive programming and debugging.

In a classical Forth assembler, the op-code itself is a Forth word that assembles the instruction according to operands passed on the stack giving the addressing information. This leads to a format in which the addressing mode specifiers precede the op-code (consistent with the postfix notation used elsewhere in Forth). Moore also standardized notation for addressing modes, although he usually used the manufacturer's instruction mnemonics. Registers were generally referred to by number, except for registers assigned to key internal system functions. For example, the stack pointer is usually in a register called **S**. One would address the second item on a two-byte wide stack using the phrase **2 S)** .

Forth assemblers support structured programming in the same way that high-level Forth does. Arbitrary branching to labeled locations is discouraged; on the other hand, structures such as **BEGIN ... UNTIL** and **IF ... ELSE ... THEN** are available in the assembler (implemented as macros that assemble appropriate conditional and unconditional branches). Such structures are easy to implement because the stack is available during assembly to carry addressing information.

Conventional assemblers leave the code in a file, which must be integrated with code in files from high-level language compilers (if any) by a linker, before the resultant program can be loaded into memory for testing. The resident Forth assembler assembles the code directly into memory in executable form, thus avoiding the linking step.

The Forth assembler is used to write short, named routines that function just like high-level Forth words: when the name of the routine is invoked, it will be executed. Like other Forth routines, code routines expect their arguments on the stack and leave their results there. Within code, a programmer may refer to constants (to get a value), variables (to get an address) or other defined data types. Code routines may be called from high-level definitions just as other Forth words, but do not themselves call high-level or code definitions.

These features enable Forth programmers to write code in short, easily testable modules that are automatically integrated into an application. Programming is fully structured, with consistent rules of usage and user interface for both assembler and high-level programming. Words are tested incrementally, while the desired behavior is fresh in the programmer's mind. Most new words can be tested simply by placing input values on the stack, typing the word to be tested and validating the result left on the stack by displaying it.

The result is complete control of the computer, high performance where needed, and overall shortening of development time due to interactive programming at all levels.

Disk Support: Classical Forth divides mass storage into "blocks" of 1024 bytes each. The block size was chosen as a convenient standard across disks whose sector sizes vary. At least two block buffers are maintained in memory, and the block management algorithm makes it appear that all blocks are in memory at all times. The command **n BLOCK** returns the memory address of block *n*, having read it if necessary. A buffer, the contents of which are changed, is marked so that when it needs to be reused, its block is automatically written out. This algorithm provides a convenient form of virtual memory for data and source storage, with a minimum number of physical disk accesses required. FORTH, Inc.'s database applications build data files out of blocks, with a file defined as spanning a specified range of blocks; data access is through operations performed against named fields within selected files.

In native Forths, the block system is both fast and reliable, as the disk driver computes the physical address of the block from its number—no directory is required. In disk-intensive applications, performance can be enhanced by adding more buffers, so more blocks will be found in memory; the buffers become a disk cache.

In the 1980s, Forth systems became available running under conventional OSs, as we shall see. Many of these support blocks within host OS files, although some have abandoned blocks altogether. As blocks provide a compatible means of accessing mass storage across both native and non-native systems, ANS Forth (Section 13.5.1) requires that blocks be available if any mass storage support is available.

Multiprogramming: The earliest Forth systems supported multiprogramming, in that the computer could execute multiple concurrent program sequences. In 1973, Moore extended this capability to support multiple users, each with a terminal and independent subdictionaries and stacks. The entity executing one of these program sequences or supporting a user is referred to as a task. Many of today's Forths support multiprogramming, and most of these use variants of Moore's approach.

This approach allocates CPU time using a cooperative, nonpreemptive algorithm: a task relinquishes the CPU while awaiting completion of an I/O operation or, upon use of the word **PAUSE**, which relinquishes the CPU for exactly one lap around the round-robin task queue.

Moore's systems used interrupts for I/O. Interrupts were directly vectored to the response code using an assembler macro, without intervention by the Forth executive. Interrupt code performed only the most time-critical operations (e.g., read a number, increment a counter), then reenabled the task

TABLE 13.2

Performance comparisons of several real-time OSs on a M68010 [Cox 1987]. Times are averages, given in μs. Times were normalized to a 10 MHz 68010. polyFORTH's use of nonpreemptive task scheduling accounts for its performance advantage.

Event:	VRTX	OS9	PDOS	polyFORTH
Interrupt response	91	43.75	93.4	7.0
Context switch	128	186.25	93.4	36
Suspend task	180	316.25	184.7	6.8
Copy memory (80 bytes)		212.5		97

that had been suspended pending the interrupt. The task would actually resume operation the next time it was encountered in the round-robin task loop, at which time it would complete any high-level processing occasioned by the event and continue its work.

In theory this nonpreemptive algorithm is vulnerable to a task monopolizing the CPU with logically or computationally intensive activity, but in practice, real-time systems are so dominated by I/O that this is rarely a problem. Where CPU-intensive operations do occur, **PAUSE** is used to "tune" performance.

Consultant Bill Cox pointed out [Cox 1987] that a nonpreemptive algorithm such as this has several advantages. First, the task scheduler itself is simpler and faster, taking as little as one machine instruction per task. Second, inasmuch as a task is suspended only at known, well-defined times, it has less "context" to be saved and restored, so the context-switch itself is faster. Third, task code can be written with the knowledge of exactly when the task does or does not control the CPU, and management of shared resources is considerably simplified. Cox compared the performance of several real-time OSs; the results are given in Table 13.2.

Tasks were constructed when the system was booted, and each was given a fixed memory allocation adequate to the functions it was intended to perform. As rebooting took only a few seconds, it was easy to reconfigure a task.

Computation: Until the late 1970s, few minicomputers offered floating point arithmetic—indeed, many lacked hardware multiply and divide. From the beginning, however, Forth was used for computationally intensive work. Controlling the radio telescope, for example, required converting wanted positions from the celestial coordinates, in which astronomical objects are located, to an azimuth/elevation coordinate system once per second and interpolating intermediate positions five times per second, with data acquisition and operator activity proceeding concurrently.

Moore's approach was to build into Forth the ability to manipulate integers effectively. For example, the command ***/** multiplies two single-cell integers and divides by a third, with a double-length intermediate product. This reflects the way most multiply and divide machine instructions work, and enables calculations such as:

```
12345 355 113 */
```

This phrase multiplies 12345 by the ratio 355/113, which represents π with an error of 8.5×10^{-8} [Brodie 1981]. The ability to multiply by a ratio is ideal for calibration and scaling, as well as rational approximations. Similarly, the word **/MOD** performs a single division, returning both the quotient and remainder. A rich set of single, double, and mixed-precision operations, such as these, make integer arithmetic much more usable than it is in most languages.

Moore expressed angles internally as 14-bit, 15-bit, or 30-bit fixed-point binary fractions. He provided a set of primitives to convert to and from angle formats (e.g., dd:mm:ss), and a math library supporting transcendental functions for these formats based largely on algorithms from Hart [1968]. Operations, such as the Fast Fourier Transform, were provided in some applications, built on specialized primitives supporting complex numbers as scaled integer pairs.

Today, fast floating-point processors are common. Many Forths support floating point, as does ANS Forth. But in many cases, such as embedded systems on simple microcontrollers, Forth's integer arithmetic still provides simpler, faster solutions.

Data Types: Perhaps, nowhere was Moore's personal philosophy more in evidence than in his approach to data typing. Basically, he wanted to assume full responsibility for manipulating data objects in whatever way he wished. If pressed on this point, he would say, "If I want to add 1 to the letter A, it's none of the compiler's business to tell me I can't."

Standard words in Forth support single and double-precision **CONSTANT**s, which return their values on the stack, and **VARIABLE**s, which return a pointer. **CREATE** names the beginning of a data region in which space can be reserved. The pointer returned by a **CREATE**d entity can be incremented to index into an array. The nature of the values kept in constants and variables was entirely arbitrary; there is normally no explicit type checking. Strings are normally kept in memory with their length in the first byte. The address of this structure, or the address and length of the actual string, can be passed on the stack.

CONSTANT, **VARIABLE**, and **CREATE** are "defining words," that is, they define new words with characteristic behaviors. Forth also provides tools to enable the programmer to build new defining words, specifying a custom behavior both at compile time (e.g., setting up and initializing a table) and run-time (e.g., accepting an index and automatically applying it to the base address of the structure).

13.3 FORTH WITHOUT CHUCK MOORE

microFORTH was heavily marketed, and attracted a lot of attention in the late '70s. One side effect of this, was the growth of an active and enthusiastic group of hobbyists who fell in love with Forth. In their wake came new companies marketing versions of Forth in competition with FORTH, Inc. At the same time, Moore himself was becoming increasingly drawn toward hardware implementations of Forth, and less involved in software production at FORTH, Inc. (which he left in 1982 to pursue his hardware interests full time). In this section, we examine the development of Forth under these diverse new influences.

13.3.1 The Forth Interest Group

In the late 1970s, Northern California was afire with the early rumblings of the Computer Revolution. Groups of interested individuals, such as the "Home Brew Computer Club," were meeting to share interests and experiences. Magazines, such as *Radio Electronics*, published step-by-step instructions on how to build your own video display terminal, and even how to build your own microcomputer system.

Due to the high cost of memory and low level of VLSI integration, typical "homebrew" computers were very resource-constrained environments. Echoing back to the first generation computers, there was insufficient memory to concurrently support an editor, assembler, and linker. Mass storage was slow and expensive, so many homebrew systems used paper tape or audio cassette tapes for I/O.

Although some BASIC language products were available, they were typically very slow, and incapable of supporting significant programs. The stage was thus set for something else to meet the expanding needs of these hardy explorers and "early adopters."

Forth had been born and bred to exploit the minimal facilities of resource-constrained systems. It carried neither the excess baggage of a general solution, nor a requirement for an existing file, or operating system, or significant mass storage. As Forth was used to tackle more and more difficult embedded computer applications, it started to claim the attention of the Northern California homebrew computer enthusiasts.

Bill Ragsdale, a successful Bay Area security system manufacturer, became aware of the benefits of microFORTH, and in 1978 asked FORTH, Inc. to produce a version of microFORTH for the 6502. FORTH, Inc. declined, seeing much less market demand for microFORTH on the 6502 than the more popular 8080, Z80 and 6800 CPUs.

Ragsdale then looked for someone with the knowledge of microFORTH and intimate familiarity with the 6502 to port a version of microFORTH to the 6502. He found Maj. Robert Selzer, who had used microFORTH for an AMI 6800 development system on an Army project and was privately developing a stand-alone editor/assembler/linker package for the 6502. Selzer wrote a 6502 Forth assembler, and used the Army's microFORTH metacompiler to target compile the first 6502 stand-alone Forth for the Jolt single board computer.

Selzer and Ragsdale subsequently made substantial modifications and improvements to the model, including exploitation of page zero and stack-implicit addressing architectural features in the 6502. Many of the enhancements that characterized the later public-domain versions were made during this period, including variable-length name fields and modifications to the dictionary linked-list threading. A metacompiler on the Jolt could target a significantly changed kernel to a higher address in memory. A replacement bootable image would then be recompiled by the new kernel into the lower boot address, which could then be written out to disk. At this point, Ragsdale had a system with which to meet his professional needs for embedded security systems.

During this period, the Forth Interest Group (FIG) was started by Ragsdale, Kim Harris, John James, David Boulton, Dave Bengel, Tom Olsen, and Dave Wyland [FIG 1978]. They introduced the concept of a "FIG Forth Model," a publicly available Forth system that could be implemented on popular computer architectures.

The FIG Forth Model was derived from Ragsdale's 6502 system. In order to simplify publication and rapid implementation across a wide variety of architectures, a translator was written to convert Forth metacompiler source code into text that, when input into a standard 6502 assembler, would replicate the original kernel image. In this way, neither the metacompiler nor its source code needed to be published. This is an important point. Forth metacompilation is a difficult process to understand completely. It requires the direct manipulation of three distinct execution phases and object areas, and is not something that a casual user wanted or needed.

By publishing assembler listings, the Forth Interest Group was able to encapsulate a Forth run-time environment in a manner that could be easily replicated and/or translated into the assembly language of a different computer architecture. It was the intention of the original team of implementors to thus stimulate the development of compatible Forth systems and the appearance of new vendors of Forth products.

After the 6502 FIG Model was published, FIG implementors published compatible versions for the 8080 and 6800 microcomputers and the PDP-11 and Computer Automation minicomputers. Over the years, volunteers added other platforms and documentation. The 1982 *Forth Encyclopedia* by Mitch Derick and Linda Baker [Derick 1982] provided an exhaustive 333-page manual on FIG Forth, with flow charts of most words. In 1983, an ad in *Forth Dimensions*, the FIG newsletter [FIG 1983],

listed: RCA 1802, 8080, PACE, 6502, 8086/88, 6800, 6809, 9900, Nova, Eclipse, VAX, Alpha Micro, Apple II, 68000, PDP11/LSI11 and Z80.

Today, there are several thousand members of the Forth Interest Group in over fifteen countries. Since 1980, FIG has sponsored an annual conference called FORML (Forth Modification Laboratory), an educational forum for sharing and discussing new or unproven proposals intended to benefit Forth, and for discussion of technical aspects of Forth. Its proceedings are available from the Forth Interest Group (P. O. Box 8231, San Jose, CA 95155).

13.3.2 Commercial and Public Domain Systems for Personal Computers

Apple Computer grew out of the bubbling computer enthusiasm in the San Francisco Bay area, and with it, a whole new generation of resource-constrained computers. Although BASIC was available in ROM, Forth was used to write a number of popular text editors and games on the Apple II, allowing resident development of significant programs within its scarce memory and disk constraints. It is hard now, with ubiquitous megabytes of memory and disk, to imagine what it was like to develop significant programs on a 40-column wide screen within 16K of memory and 100K of disk storage.

Vendors of low cost Forth systems sprang up almost overnight, each supporting their favorite personal computer, most of them basing their systems on the FIG model. In 1979, for example, Miller Microcomputer Services announced MMSFORTH for the TRS-80 [TRS-80 1979], and by 1980 *Computerworld* reported [Taylor 1980], that MMS had over 100 user groups for its product.

When IBM entered the personal computer business, with their original PC product offering, they chose to distribute a version of the popular Apple II text editor EasyWriter, written in Forth, as an IBM product. Laboratory Microsystems (LMI) introduced a commercial IBM PC Forth system in 1982. Numerous commercial and public domain Forth products followed, and significant software product development began.

Following its introduction of the first commercial Forth for the IBM-PC, LMI has maintained a continuing strategy of producing cutting-edge Forth systems for the PC, including a 32-bit real-mode implementation (February, 1983), an OS/2-based Forth (February, 1988) and a Windows version (1992). Along the way LMI's founder, Ray Duncan, became an acknowledged authority on Microsoft OSs [e.g., Duncan 1988].

FORTH, Inc.'s PC offering was polyFORTH, which combined the multi-user support and database tools of its minicomputer products with the ROMable architecture of microFORTH. By 1984 FORTH, Inc. was supporting up to 16 users on a PC with no visible degradation, and running polyFORTH —first as a native OS and later as a coresident OS with MS-DOS. By the late '80s polyFORTH users, such as NCR, were supporting as many as 150 users on a single 80386-based PC.

In 1978, Major Seltzer gave Don Colburn a copy of the 6502 Forth he wrote for Ragsdale in exchange for Colburn's writing two articles on Selzer's 6502 work. Colburn subsequently used this as a basis for a version based on the preliminary FORTH-77 standard (the only FORTH-77 implementation of which the authors are aware). In the Fall of 1979, Colburn generated a FIG-compatible system for prototypes of the 68000. A multitasking, multiuser version of this product called MultiForth was demonstrated to Motorola in January, 1980, well ahead of production shipments of the 68000. When Hewlett Packard's desktop-computer division designed a new generation of desktop computers around the 68000 in 1982, the first available third-party language product they distributed under an HP part number was MultiForth.

Colburn's company "Creative Solutions" also introduced MacForth, the first resident development system for the 128K Apple Macintosh, immediately after the Mac's debut in January of 1984. Because MacForth uniquely provided direct access to the entire Macintosh "Toolbox ROM" (routines in a

TABLE 13.3

Some major suppliers of Forth systems, services and related products.

System(s)	Company	Primary Products & Markets
CFORTH83, Forthmacs, SunForth	Bradley Forthware	Portable Forth written in C; versions for Atari, Macintosh, Sun; consulting and services related to the Sun Microsystems Open Boot.
cmFORTH	Silicon Composers and others	Public-domain system for Novix Forth processor by C. Moore, ported to the Harris and SC-32 Forth processors by others.
Cyrano	Opto-22	Forth for a proprietary embedded controller
F-PC	T. Zimmer et al.	Extensive public-domain system for the IBM-PC family
F83	Laxen and Perry	Public-domain system for the IBM-PC family, later ported by others to other platforms
HS/Forth	Harvard Softworks	IBM-PC family
JForth	Delta Research	Amiga
MacForth	Creative Solutions, Inc.	Apple Macintosh, NuBus interface boards
Mach2	Palo Alto Shipping	Apple Macintosh
mmsFORTH	Miller Microcomputer Services	IBM-PC family; business and commercial applications
MPEForth	MicroProcessor Engineering (UK)	PCs and embedded systems
mvpFORTH	Mountain View Press	Public-domain system on a variety of platforms
Open Boot	Sun Microsystems	Programmable ROM-based Forth on SPARC workstations
polyFORTH	FORTH, Inc.	Industrial systems on PCs and other platforms; interactive cross-compilers; consulting and custom programming services
UR/Forth	Laboratory Microsystems, Inc. (LMI)	IBM-PC family running DOS, OS2 and Windows; also cross compilers for a variety of systems

resident programming environment, along with comprehensive application examples), a majority of the first generation of Macintosh application programmers learned how to create, and use, pull-down menus, windows, graphics and mice with MacForth. Significant large-volume spreadsheets, 2D and 3D rendering and design packages, CAD/CAM design tools, games, medical diagnostics, image enhancement programs, accounting packages, desktop planetariums, and process control applications were written on early Macintoshes in MacForth.

Byte Magazine dedicated its August, 1980 issue to Forth. It was their largest-selling issue to date, and was reprinted several times.

By 1985, there were over 70 vendors of Forth systems, ranging from single individuals to multimillion dollar organizations.

In 1982, Lawrence Forsley founded the Institute for Applied Forth Research, now called simply the Forth Institute. This organization sponsors an annual Conference on Forth Applications at the University of Rochester, Rochester, NY, and publishes the *Journal of Forth Application and Research*, a refereed technical periodical on applications of Forth, new developments and techniques, and surveys of specific areas of Forth.

In 1989, George Shaw and others formed an ACM Special Interest Group on Forth called SIGForth, which also sponsors a newsletter and an annual conference.

13.3.2.1 *Design principles*

FIG Forth was optimized for portability rather than performance. Only a very few primitives were coded in assembler, and the rest of the logic was implemented using high-level Forth. As a result, it was fairly slow—some operations, such as dictionary searches, were a factor of ten slower than representative commercial implementations.

Other internal decisions were similarly made with the neophyte in mind. For example, the earlier FORTH, Inc. systems compiled word names as the length of the name and the first three characters. This gave a lower collision rate than simple truncation, and was adequate most of the time. But, the FIG model used variable-length names up to 31 characters, thereby trading size for user-friendliness. This was somewhat controversial at the time (see Fig. 13.1), but by the mid 1980s, most systems had converted to this usage.

The advent of personal computers provided Forth implementors with the incentive to learn to run under a host OS. The first non-native systems were developed in 1980 by Martin Tracy of Micromotion (for the Apple II) and Ray Duncan of Laboratory Microsystems, Inc. (for CP/M on Z80s). LMI's system, also featured a full-screen editor. In 1981, LMI added support for a software and hardware floating point, and also pioneered performance enhancements, such as native code translation and caching dictionary lookups in a hash table to accelerate dictionary searches.

The advent of non-native Forth implementations introduced an issue that remains controversial in Forth practice today, the use of host OS files for mass storage. There are two main approaches: abandoning traditional blocks altogether in favor of directly manipulating source and data in files, and mapping blocks to host OS files. The former approach is favored by implementors who are concentrating on systems for a particular OS (e.g., MS-DOS), whereas the latter is preferred by organizations such as FORTH, Inc. that support both native and non-native products.

FIGURE 13.1

"Letter to the Editor" of *Forth Dimensions* [Moore 1983] concerning the practice of storing names of Forth words as a count and first three characters.

```
DEA- EDI---

I AM AFR--- THA- THE LET--- IN THE LAS- ISS-- ABO--
FOR-- INC- USI-- ONL- THR-- LET--- NAM- FIE--- HAS
HAD THE OPP----- EFF--- FRO- WHA- THE WRI--- WAN---

HIS LET--- ( LIK- THI- ONE ) SHO-- THA- SAV--- ONL-
THR-- LET---- AND COU-- IS JUS- ABO-- OPT---- IN
TER-- OF A TRA-- OFF BET---- SAV--- MEM--- AND
KEE---- LEG-------

WE STI-- DON-- SEE THE NEE- FOR 31 CHA------ NAM--
IN THE GEN---- CAS-

YOU-- TRU--

CHU-- MOO--

FOR-- INC-
```

Creative Solutions' MacForth used very compact object image strategies, including token threading and separated name heads to maximize the amount of memory available for program development on the original 128K Macintoshes. Other novel features included run-time relocation of the executable image and exclusion of word names in run-time systems without metacompilation. MacForth included a seamless programming environment, incorporating screen based text editor, compiler, interpreter, and assembler in under 20k bytes of memory.

13.3.2.2 *Influences*

The FIG Model was in the public domain, and was ported to a wide variety of computer systems. Because the internal design of FIG Forth was essentially the same across all machines, programs written in FIG Forth enjoyed a substantial degree of portability, even for "system-level" programs that directly manipulated the internals of dictionary entries and other implementation-dependent features. Because FIG Forth was the first introduction to Forth for many people, it is widely associated with "the nature of Forth."

However, FIG Forth was not representative of all commercial implementations of this era. Commercial vendors tended to be much more performance-conscious, and elected implementation strategies that optimized performance or size rather than porting ease, as we have seen.

The first major effort to standardize Forth was at a meeting in Utrecht in 1977, attended by several astronomical Forth users and FORTH, Inc. (at that time the only commercial vendor). They produced a preliminary standard called FORTH-77, and agreed to meet the following year. Meetings in 1978 and 1979 on Catalina Island in California, now including representatives from the Forth Interest Group and other producers, yielded a more comprehensive standard called FORTH-79. Although FORTH-79 was very influential, many Forth users and vendors found flaws in it; in 1982 two meetings were held to update the standard, and in 1983 a new standard was released called FORTH-83. Both FORTH-79 and FORTH-83 specified a 16-bit, twos-complement, unaligned, linear byte-addressed virtual machine, and included a number of assumptions about implementation techniques.

Unfortunately, some of the changes in FORTH-83 produced grave incompatibilities with existing code. For example, the formal representation of a "true" flag had always been 1, and the word **NOT** inverted a Boolean flag. In FORTH-83, "true" became –1 and **NOT** became a bit-wise complement. Other problems involved the specification for floored division in FORTH-83 and a serious ambiguity in the specification of parameters for certain loop structures. The effect of these incompatibilities was divisive. Although most implementors agreed that FORTH-83 was an improvement and adopted the new standard, there remains a vocal group who never converted, and who remain skeptical of the whole standards process. Of the systems listed in Table 13.3, for example, most are fairly close to FORTH-83 compatibility; notable exceptions are MacForth, mmsFORTH and mvpFORTH, all of which stayed with FORTH-79.

In 1981, Prentice Hall published *Starting FORTH*, by Leo Brodie [Brodie 1981], then an employee of FORTH, Inc. Both lucid and entertaining (Brodie drew memorable cartoon figures representing important Forth primitives), *Starting FORTH* was also a thorough introduction to the language. It sold over 110,000 copies (for a time it was the best-seller in Prentice Hall's computer line) and exerted a powerful influence on many people learning about Forth for the first time, as well as on vendors scrambling to be compatible with it. Although the first edition was primarily based on FORTH, Inc.'s polyFORTH, it included many footnotes and examples in FIG Forth and other dialects. The second edition (1987) was based on the FORTH-83 standard.

Another major influence in the personal computer marketplace has been the competition between public-domain and commercial versions of Forth. In the mid-1980s, the FIG model was gradually

replaced by the public domain F83 (produced by Henry Laxen, Mike Perry, and others, operating under the name "No Visible Support Software"), a multitasking system originally released on the IBM-PC. Versions have been developed by many independent programmers on a wide variety of other platforms. This system is so widespread that many people are led by its name to confuse it with the FORTH-83 standard. In fact, although it is largely compatible with FORTH-83, F83 goes well beyond the limited FORTH-83 standard in its features. In the late '80s, Tom Zimmer and others produced an even more extensive public-domain system for PCs called F-PC, which includes several megabytes of source code and utilities. But, except for these, most public-domain Forths are rather limited.

Public-domain Forths have certainly helped to ensure that Forth is widely known. But their influence isn't entirely benign. According to Tyler Sperry, editor of *Embedded Systems Programming Magazine* [Sperry 1991]:

> The problem is that it is relatively easy to implement your own minimal Forth system. The kernel, after all, is only a few hundred bytes of code.... Unfortunately, bringing up a Forth interpreter is like writing a Small C compiler: it's only a toy without a well-developed library. One of the biggest problems with public-domain and shareware systems is that their libraries are often only partially completed, with sketchy documentation. And that's putting the situation kindly.

People who have only seen or used limited public-domain Forth implementations often perceive that Forth itself is a toy. And suppliers of high-quality commercial systems must deal with prospective customers' assumptions that all Forths are the same, an assumption that naturally creates considerable price resistance, given that the public-domain versions are extremely inexpensive. The standing joke within the Forth community, however, is "when you've seen one Forth ... you've seen one Forth." The range in quality of code and documentation, nature and extent of libraries, as well as product support, is enormous. A prospective user is well advised to evaluate a number of both public domain and commercial offerings.

13.3.3 Embedded Systems

13.3.3.1 *Environment and applications*

Forth's ability to make maximum use of limited hardware resources made it a natural choice for embedded uses of microprocessors. Some of these have been small: an RCA 1802-based cardiac monitor (1979) that performed a detailed waveform analysis of heart beats was not much larger than the 1" x 2" tape cassette it used to record abnormalities. Some were large, such as the 750-ton stretch press used by Lockheed to form panels for the C5B airplane wings in the early '80s. Some were distributed, such as the roughly 500 networked processors used for an extensive facility management system at the King Khaled International Airport at Ryadh, Saudi Arabia [Rather 1985]. Forth has been especially successful in developing firmware for hand-held devices made by companies such as Itron and MSI Data. In 1990, Federal Express won the prestigious Malcolm Baldridge quality award for its package-tracking system, data entry of which is performed by Forth-based hand-held devices carried by Federal's 50,000 couriers and agents world-wide.

Forth's extreme modularity facilitates thorough, systematic testing, which has made it attractive for applications requiring high reliability. As a result, it has been used in a number of satellites and Space Shuttle experiments. McDonnell Douglas used polyFORTH in their Electrophoresis in Space project [Wood 1986] to control the cargo bay factory itself (multiple 68000 VME-bus boards), the astronaut's control console (a laptop PC), and their ground-based analysis computer (a Compaq PC).

The November, 1990 Columbia shuttle flight carried four astronomy payloads, of which three were programmed in Forth [Ballard 1991], and the January, 1992, Spacelab flight featured a Microgravity Vestibular Investigation (MVI) experiment using a polyFORTH system for on-board control and analysis [Paloski 1986] and MACH2 in a ground-based Macintosh for analysis.

Probably the most prolific single purveyor of embedded Forths is Sun Microsystems, whose SPARC workstations all use a programmable Forth-based monitor called Open Boot, developed by Mitch Bradley and associates. Bradley believes [Bradley 1991] that Forth was successful for this purpose because it offered:

1. a CPU-independent "virtual machine" to use for the byte-coded portable drivers;
2. a debugging environment for those drivers;
3. an interactive command language, with complete programming language capability, that was useful for hardware startup and debugging;
4. a built-in debugging environment for the firmware itself (firmware is otherwise rather painful to debug);
5. a debugging environment for the operating system software;
6. extensibility, allowing easy support of new hardware requirements and features; and
7. great flexibility in tuning the implementation for speed/space tradeoffs.

At least one other major board-level CPU vendor has adopted Open Boot firmware across their product line, and there is a working group developing an IEEE standard for it.

13.3.3.2 *Design principles*

From the minis of the '70s to the PCs of the '80s, most Forth systems have supported development on the same computer on which the completed application is to run. Even the microprocessor systems of the late '70s and early '80s were developed on the same CPU (as opposed to cross-development), with development software features for stripping the development tools and producing a ROMable target.

Most embedded systems lack a disk, a terminal, or both, thereby rendering themselves inhospitable to even the leanest Forth programming environment. Nonetheless, some vendors do provide on-board Forths in microcontrollers. Examples include the Rockwell AIM 65 mentioned before, and microcontroller boards sold by New Micros, Inc. of Texas; Vesta Technologies, Inc., in Colorado; and Opto-22 in California.

But as PCs became ubiquitous, they also became popular as hosts for more comfortable and powerful Forth cross-development environments. These have generally been based on modified versions of the classical Forth metacompilers, adapted to support cross development.

The traditional Forth dictionary is integrated: a "definition" includes the word's name (which can be found in a dictionary search performed by the text interpreter), an executable portion (typically a pointer to code that executes words of a particular class, such as colon definitions, variables, constants, etc.), and data space (containing one or more values or addresses of words that make up the content of the definition), all classically in contiguous memory locations. (However, see Section 13.5.2, Implementation Strategies). A metacompiler divides these structurally into portions that are used by the host system's compiler (equivalent to a symbol table) and portions required at run-time in the target. In order for a target program to be ROMable, the compiler must also manage separate ROM and RAM data spaces, usually using multiple sets of dictionary pointers.

13.4 HARDWARE IMPLEMENTATIONS OF FORTH

The internal architecture of Forth simulates a computer with two stacks, a set of registers, and other well-defined features. As a result, it was almost inevitable that someone would attempt to build a hardware representation of the actual Forth computer.

The first such effort was made in 1973 by John Davies of the Jodrell Bank Radio Astronomy Observatory near Manchester, England. Davies' approach was to re-design a Ferranti computer that had gone out of production to optimize its instruction set for Forth.

The first actual Forth computers were bit-sliced board-level products. The first of these was made by a California company called Standard Logic, in 1976. By making a minor modification in the instruction set of their board-level computer, Standard Logic's chief programmer Dean Sanderson was able to implement the precise instruction that Forth uses in its "address interpreter" to move from one high-level command to the next. Their system was used widely by the US Postal System.

In the early 1980s, Rockwell produced a microprocessor with Forth primitives in on-chip ROM, the Rockwell AIM 65F11 [Dumse 1984]. This chip has been used quite successfully in embedded microprocessor applications. However, no attempt was made to adapt the actual architecture of the processor (basically a 6502) for Forth support.

In 1981, Moore himself undertook design of an actual Forth chip. Working first at FORTH, Inc. and subsequently with a start-up company called Novix, formed to develop the chip, Moore completed the design in 1984, and the first prototypes were produced in early 1985 [Golden 1985]. This design was subsequently purchased and adapted by Harris Semiconductor Corp., and formed the basis of their line of RTX processors.

Starting in the early 1980s, a group at the John Hopkins Applied Physics Laboratory in Maryland developed a series of experimental Forth processors for use in space instrumentation [Hayes 1987]. The most successful of these, marketed as the SC-32 by Silicon Composers of Palo Alto, CA, was used to control the Hopkins Ultraviolet Telescope which flew in the Columbia Space Shuttle in November, 1990 [Ballard 1991]. It continues to be the basis for more space instruments under development.

Moore himself, working on his own, has continued to develop Forth-based processors for special applications.

The various Forth processors have had an influence on Forth software systems. In order to take full advantage of these architectures, Forth compilers were developed by Moore, FORTH, Inc., and Laboratory Microsystems that generated machine code optimized for the chip's internal architecture. A native looping structure in the Novix and Harris chips called **FOR ... NEXT** (which counted down from a single-argument upper limit to zero) led to adoption of this structure in other Forths as well.

13.5 PRESENT AND FUTURE DIRECTIONS

The computer industry has always been characterized by rapid and profound changes. Because Forth was last standardized in the early 1980s, the speed, memory size, and disk capacity of affordable personal computers have increased by factors of more than one hundred. 8-bit processors are now rare in PCs (although they are still widely used in embedded systems), and 32-bit processors are common. Operating systems, programming environments, and user interfaces are far more sophisticated. Many recent Forth implementations, both commercial and public-domain, have attempted to address these issues.

13.5.1 Standardization Efforts

At the time of writing (November, 1992), a Technical Committee X3J14 (of which authors Rather and Colburn are members) is nearing completion of an ANS Forth. Among the 20 voting members in the TC are vendors (FORTH, Inc., Creative Solutions, Sun Microsystems, and a division of NCR), some large user organizations (Ford Motor Co., NASA), and a number of smaller user organizations, consultants and experts. Starting in 1987, this group has addressed a number of problems with FORTH-79 and FORTH-83, as well as some contemporary issues. A few of the issues addressed in the draft standard follow, as they represent current areas of lively debate and technical activity among Forth users and implementors.

ANS Forth attempts to reconcile some of the divisions caused by the incompatibilities between FORTH-79 and FORTH-83. For example, it retains **0=** to perform the FORTH-79 **NOT** function, introduces **INVERT** to perform the FORTH-83 **NOT**, and removes the word **NOT**. This enables application writers who depend on either version to leave their programs unchanged, and achieve compatibility by adding a simple shell in which **NOT** is defined as a synonym for the preferred behavior.

The proposed standard also removes virtually all restrictions on implementation options, provides for independence from CPU word size, and offers a number of optional extension word-sets for functions such as host OS file compatibility, dynamic memory allocation, and floating point arithmetic. Some significant issues addressed by ANS Forth follow.

13.5.1.1 *Cell size*

FORTH-79 and FORTH-83 mandated a 16-bit architecture, including stack width, addresses, flags, and numbers. ANS Forth specifies sizes in terms of a "cell," the width of which is implementation-defined but must be at least 16 bits. Words have been added to increment addresses transportably by a cell, a character, or an integral number of cells or characters.

13.5.1.2 *Arithmetic*

Amid great controversy, FORTH-83 mandated floored division. Not only was this incompatible with prior usage (which didn't specify the algorithm for handling signed division), it was also at variance with hardware multiply/divide instructions on most processors. But many people felt strongly that floored division is mathematically more appropriate, and that it was important to specify. Recognizing that there were many implementations on both sides of this issue, the TC opted to allow either floored or truncated division. The implementation must specify which default it uses, and must provide primitives supporting both methods.

13.5.1.3 *Control structures*

One of the unique characteristics of Forth is the degree to which its own internal tools are accessible to the application programmer. For example, there is one lexical analyzer used by the compiler, assembler, and text interpreter; it is also available for command and text parsing in applications. Similarly, the tools that implement control structures, such as loops and conditionals, are available for making custom structure words. In 1986, Wil Baden demonstrated [Baden 1986] that the standard Forth structure words, plus a few extensions made from these underlying tools, are adequate to make any structure, including solutions to problems posed in D. E. Knuth's paper "Structured Programming with **go to** statements" [Knuth 1974].

TABLE 13.4

Standard control structures in Forth. ANS Forth allows programmers to form new structures by mixing the component words or using them to define new structure words.

Structure	Description
`DO ... LOOP`	Finite loop incrementing by 1
`DO ... <n> +LOOP`	Finite loop incrementing by `<n>`.
`BEGIN ... <f> UNTIL`	Indefinite loop terminating when `<f>` is 'true'
`BEGIN ... <f> WHILE ... REPEAT`	Indefinite loop terminating when `<f>` is 'false'
`BEGIN ... AGAIN`	Infinite loop
`<f> IF ... ELSE ... THEN`	Two-branch conditional; performs words following **IF** if `<f>` is 'true' and words following **ELSE** if it is 'false'. **THEN** marks the point at which the paths merge.
`<f> IF ... THEN`	Like the two-branch conditional, but with only a 'true' clause.

FORTH-79 and FORTH-83 provided syntactic specifications for the common structures listed in Table 13.4, as well as an "experimental" collection of structure primitives. The latter were not widely adopted, however, and few implementations perform the kind of syntax checking the standards anticipated. F83 offers a limited form of syntax checking, in that it requires the stack, which is used at compile-time for compiling structures, to have the same size before and after compiling a definition, the theory being that a stack imbalance would indicate an incomplete structure. Unfortunately, this technique prevents the very common practice of leaving a value on the compile-time stack which is to be compiled as a literal inside a definition.

Common practice often took advantage of knowledge about how the structure words worked at compile-time to manipulate them in creative ways. The ANS Forth Technical Committee sanctioned this by providing specifications of both the compile-time and run-time behaviors of the structure words, so that they may be combined in arbitrary order. A set of structure primitives is provided in a "programming tools" wordset, and the word **POSTPONE** is provided to enable programmers to write new structure words that reference existing compiler directives, in order to provide a portion of the desired new behavior.

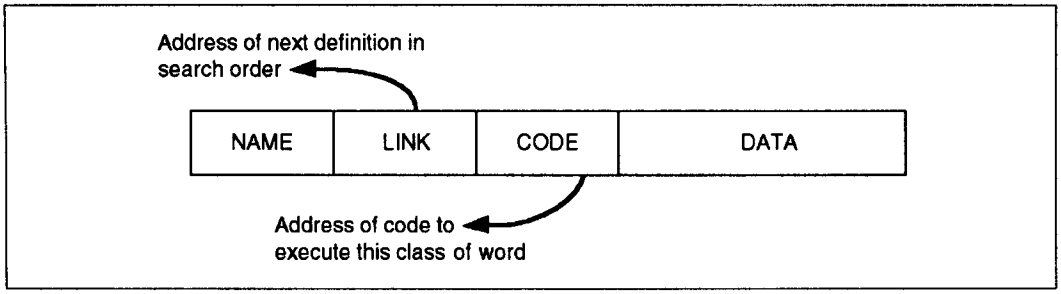
13.5.2 Implementation Strategies

The original Forth systems, developed by Moore in the 1970s, compiled source from disk into an executable form in memory. This avoided the separate compile-link-load sequences characteristic of most compiled languages, and led to a very interactive programming style in which the programmer could use the resident Forth editor to modify source and recompile it, having it available for testing in seconds. The internal structure of a definition was as shown in Fig. 13.2, with all fields contiguous in memory. The FIG model and its derivatives modified the details of this structure somewhat, but preserved its essential character.

Forth systems implemented according to this model built a high-level definition by compiling pointers to previously defined words into its parameter field; the address interpreter, that executed such definitions, proceeded through these routines, executing the referenced definitions in turn by performing indirect jumps through the register used to keep its place. This is generally referred to as *indirect-threaded* code.

FIGURE 13.2

Diagram showing the logical components of a Forth definition. In classical implementations, these fields are contiguous in memory. The data field will hold values for data objects, addresses, or tokens for procedures, and the actual code for **CODE** definitions.



The need to optimize, for different conditions, has led to a number of variants in this basic implementation strategy, however. Some of the most interesting are:

1. **Direct threaded code.** In this model, the code field contains machine code instead of a pointer-to-machine code. This is somewhat faster, but typically costs extra bytes for some classes of words. It is most prevalent on 32-bit systems.
2. **Subroutine-threaded code.** In this model, the compiler places a jump-to-subroutine instruction with the destination address in-line. This technique costs extra bytes for each compiled reference on a 16-bit system. It is often slower than direct-threaded code, but it is an enabling technique to allow the progression to native code generation.
3. **Native code generation.** Going one step beyond subroutine-threaded code, this technique generates in-line machine instructions for simple primitives such as + and jumps to other high-level routines. The result can run much faster, at some cost in size and compiler complexity. Native code can be more difficult to debug than threaded code. This technique is characteristic of optimized systems for the Forth chips such as the RTX, and on 32-bit systems where code compactness is often less critical than speed.
4. **Optimizing compilers.** A variant of native code generation, these were invented for the Forth processors that can execute several Forth primitives in a single cycle. They looked for the patterns that could be handled in this way and automatically generated the appropriate instruction. The range of optimization was governed by the capabilities of the processor; for example, the polyFORTH compiler for the Novix and RTX processors had a four-element peephole window.
5. **Token threading.** This technique compiles references to other words using a token, such as an index into a table, which is more compact than an absolute address. Token threading was used in a version of Forth for a Panasonic hand-held computer developed in the early 1980s, for example, and is a key element in MacForth.
6. **Segmented architectures.** The 80x86 family supports segmented address spaces. Some Forths take advantage of this to enable a 16-bit system to support programs larger than 64K. Similarly, implementations for Harvard-architecture processors such as the 8051 and TI TMS320 series manage separate code and data spaces.

Although the early standards assumed the classical structure, ANS Forth makes a special effort to avoid assumptions about implementation techniques, resulting in prohibitions against assuming a

relationship between the head and data space of a definition or accessing the body of a data structure other than by predefined operators. This has generated some controversy among programmers who prefer the freedom to make such assumptions over the optimizations that are possible with alternative implementation strategies.

13.5.3 Object-Oriented Extensions

Forth's support for custom data types with user-defined structure, as well as compile-time and run-time behaviors, has, over the years, led programmers to develop object-based systems such as Moore's approach to image processing described in Section 13.2.2.2, item 2. Pountain [1987] described one approach to object-oriented programming in Forth, which has been tried by a number of implementors. Several Forth vendors have taken other approaches to implementing object-based systems, and this is currently one of the most fertile areas of exploration in Forth.

In 1984, Charles Duff introduced an object-oriented system written in Forth called Neon [Duff 1984 a & b]. When Duff discontinued supporting it in the late '80s, it was taken over by Bob Lowenstein, of the University of Chicago's Yerkes Observatory, where it is available as a public-domain system under the name Yerk. More recently, Michael Hore re-implemented Neon using a subroutine-threaded code; the result is available (also in the public domain) under the name MOPS. Both Yerk and MOPS are available as down-loadable files on a number of Forth-oriented electronic bulletin boards listed at the end of this paper.

13.6 *A POSTERIORI* EVALUATION

The early development of FORTH was, in many ways, quite different from that of most other programming languages. Whereas they generally emerged full featured, with unambiguous formal specifications for language syntax and semantics, Forth enjoyed a lengthy, dynamic adolescence, in which each fundamental presupposition of the language was tested on the anvil of actual applications experience. During this period, Moore, unencumbered by a large following of users, often made revolutionary changes to the language on a daily basis, to suit his current view of what the language should be. He had complete control and responsibility for the machine at hand, from the first bootstrap loader to the completed application. The language converged toward the actual needs of one man solving a broad class of technically challenging problems in resource-constrained environments.

The resulting method of problem solving, expressed by the resulting *de facto* language specification, has proven useful to others. Given the complete flexibility to add syntax checking, data typing, and other more formal structures often considered essential to programming languages, most of the several hundred people who have independently implemented versions of Forth for their own use have not done so. The results of their efforts, as surveyed by the ANS Forth Technical Committee, represent a startlingly democratic ratification of Moore's personal vision.

13.6.1 Meeting Objectives

Without a formal language design specification citing clearly defined objectives, we can only evaluate the stated objectives of the inventor of the language and those who have used it. Personal productivity and intellectual portability were Moore's primary stated objectives. Forth has been ported across the vast majority of programmable computers and has been embodied in several different dedicated Forth computer architectures.

In 1979, Chuck Moore looked back on ten years' experience with Forth and observed [Moore 1979]:

> My original goal was to write more than 40 programs in my life. I think I have increased my throughput by a factor of 10. I don't think that throughput is program-language limited any longer, so I have accomplished what I set out to do. I have a tool that is very effective in my hands—it seems that it is very effective in others' hands as well. I am happy and proud that this is true.

Today he sees no reason to change this assessment.

The developers of FIG Forth saw their systems spread all over the world, along with chapters of their organization, and influence Forth programmers everywhere. Their goal of instantiating additional commercial vendors of Forth products was also achieved.

Of the many entrepreneurs who committed their careers and fortunes to Forth-based enterprises, few have become rich and famous for their efforts. But most have had the satisfaction of seeing their own productivity increased just as Moore did, and of having seen seemingly impossible project objectives met because of the power and flexibility of the language. They have also enjoyed prosperity in making this capability available to their clients and customers.

Given Moore's criteria of productivity and portability, perhaps the best measure of achieving these objectives is the very large quantity and range of application programs that have been written in Forth by a small number of programmers across a very broad variety of computers.

13.6.2 Major Contributions of Forth

In 1984, Leo Brodie wrote a book on designing Forth applications called *Thinking Forth* [Brodie 1984]. In it, he quoted a number of Forth programmers on their design and coding practices. In an Epilogue, several of them commented that Forth had significantly influenced their programming style in other languages, and indeed their approaches to problem solving in general. Here are two examples, which are typical of observations of Forth users in general:

> [The] essence of good Forth programming is the art of factoring procedures into useful free-standing words. The idea of the Forth word had unexpected implications for laboratory hardware design.
>
> Instead of building a big, monolithic, all-purpose Interface, I found myself building piles of simple little boxes which worked a lot like Forth words: they had a fixed set of standard inputs and outputs, they performed just one function, they were designed to connect up to each other without much effort, and they were simple enough that you could tell what a box did just by looking at its label. . . .
>
> Because Forth is small, and because Forth gives its users control over their machines, Forth lets humans control their applications. It's just silly to expect scientists to sit in front of a lab computer playing "twenty questions" with packaged software. Forth . . . lets a scientist instruct the computer instead of letting the computer instruct the scientist.
>
> — Mark Bernstein, president of Eastgate Systems, Inc., Cambridge, MA
>
> Forth has changed my thinking in many ways. Since learning Forth I've coded in other languages, including assembler, BASIC and FORTRAN. I've found that I used the same kind of decomposition we do in Forth, in the sense of creating words and grouping them together.
>
> More fundamentally, Forth has reaffirmed my faith in simplicity. Most people go out and attack problems with complicated tools. But simpler tools are available and more useful.
>
> — Jerry Boutelle, owner of Nautilus Systems, Santa Cruz, CA

Mitch Bradley reports [Bradley 1991] that the design of the Forth-based Open Boot has significantly influenced the thinking of the people at Sun Microsystems who are responsible for the low-level interfaces in the Unix kernel. Open Boot design philosophy is influencing driver interfaces, the device naming system, and the early startup and configuration mechanisms. There is even talk of unifying the syntax of several disparate kernel configuration files by using Forth syntax and including a subset Forth interpreter in the Unix kernel. People at Sun, who have worked with Open Boot, are impressed by the fact that the simple postfix syntax never "runs out of steam" or "paints you into a corner."

13.6.3 Mistakes or Desired Changes

Forth has a chameleon-like capacity to adapt to any particular application need. Indeed, the process of programming in Forth is to add to it application-oriented words at increasingly high levels until all the desired functionality is implemented. So, for any project, or even any particular programming group, any perceived needs will be promptly addressed. When looking for "mistakes" then, the most useful questions to ask are, "What were the things that a significant number of implementors have chosen to change or add?" and, "What are the characteristics of the language that may have prevented its wider acceptance?"

One of the first actions taken by the ANS Forth Technical Committee, when it formed in 1987, was to poll several hundred Forth implementors and users to determine their views on problems in the language that needed to be addressed. The issues cited fell into three categories: "mistakes" in one or both of the existing standards (e.g., incompatibilities introduced by FORTH-83 and anomalies such as an awkward specification for arguments to **DO**); obsolete restrictions in FORTH-83 (mainly the reliance on a 16-bit architecture); and a need for standards for such things as host file access, floating point arithmetic, and so forth. Features in the latter group were, by then, offered by most commercial and many public-domain systems, but as they had been developed independently, there was variance in usage and practice. ANS Forth has attempted to address all these concerns.

In retrospect, however, the lack of standard facilities for such things as floating-point arithmetic, which are covered by other languages, has probably impeded widespread acceptance of Forth. It's insufficient to point out that most commercial systems offer them, if the public perception of the language is formed by a standard that omits any mention of such features! From this perspective, the ANS Forth effort has come almost too late.

Another difficulty is that Forth's very identity is unclear: it is not only unconventional in appearance with its reliance on an overt stack architecture and postfix notation, but it broadly straddles territory conventionally occupied by not only languages, but also operating systems, editors, utilities, and the like, that most people are accustomed to viewing as independent entities. As a result, it's difficult to give a simple answer to the question of what it is.

The integrated character of Forth is viewed by its practitioners as its greatest asset. As Bradley [1991] expresses it,

> Forth has taught me that the 'firewalls' between different components of a programming environment (i.e., the different syntax used by compilers, linkers, command interpreters, etc.) are very annoying, and it is much more pleasant to have a uniform environment where you can do any thing at any level at any time, using the same syntax.

Duncan [1991], however, believes that this seamless integration of Forth the language, Forth the virtual machine, and Forth the programming environment is a significant barrier to mainstream acceptance. He notes that the same has been observed regarding Smalltalk versus C++:

> I have been using C++ for some months now, and the very things about C++ that frustrate me—the language is not written in itself (thus there is no way to use the building blocks for the programming environment as part of the application), the language is not truly extensible (e.g., the operators for the native data types cannot be overridden), and there is no programming environment that is smart about the language and class hierarchies—are the things that traditional language experts see as assets for C++ compared to Smalltalk!

As long as Forth users are convinced that its integrated, intrinsically interactive character is the key to their productivity as programmers, however, it is unlikely to change.

13.6.4 Problems

Some languages tend to be "levelers:" that is, a program written by an expert is unlikely to be significantly better (smaller, faster, and so forth) than one written by a novice. Chuck Moore once observed [Moore 1979], "...FORTH is an amplifier. A good programmer can do a fantastic job with FORTH; a bad programmer can do a disastrous one." Although never quantified, this observation has been repeated on many Forth projects across a broad programmer population, and has achieved the status of "folk wisdom" within the Forth community.

This tendency has given Forth the reputation of being "unmanageable," and there have been some highly publicized "Forth disasters" (notably Epson's VALDOCS project in the early 1980s). On close examination, however, the root causes of this, and other failed Forth projects, are the same problems that doom projects using other languages: inadequate definition, poor management, and unrealistic expectations.

There have also been a number of Forth successes, such as the facility management system for the Saudi Arabian airport mentioned before, in which a project that was estimated to contain 300,000 lines of executable FORTRAN, PLM, and assembly language software was totally redesigned, recoded in Forth, and tested to the satisfaction of the customer in only eighteen months [Rather 1985]. The result ran more than a factor of ten faster.

Jack Woehr, a senior project manager for Vesta Technologies, observes [Woehr 1991] that successful management of Forth projects demands nothing more than generally good management practices, plus, an appreciation of the special pride that Forth programmers take in their unusual productivity. Forth rewards a management style that believes a small team of highly skilled professionals can do a better job, in a shorter time, at less overall cost, than a large group of more junior programmers.

13.6.5 Implications for Current and Future Languages

What can be learned from 20 years experience with Forth? Forth stands as a living challenge to many of the assumptions guiding language developers. Its lack of rigid syntax and strong data typing, for example, are characteristically listed as major advantages by Forth programmers. The informal, interactive relationship between a Forth system and its programmer has been shown through many projects to shorten development times, in comparison with more conventional tools such as C. Despite the tremendous increases in the size and power of modern computers, Forth's combination of easy programming, compact size, and fast performance (characteristics often thought to be mutually exclusive) continues to earn a loyal following among software developers, especially for embedded systems.

REFERENCES

[ANS 1991] *Draft Proposed ANS Forth*, document number X3.215-199x, available from Global Engineering Documents, 2805 McGaw Ave., Irvine, CA, 92714.

[Baden, 1986] Baden, W., Hacking Forth, *Proceedings of the Eighth FORML Conference*, pub. by the Forth Interest Group, P. O. Box 8231, San Jose, CA 95155, 1986.

[Ballard, 1991] Ballard, B., and Hayes, J., Forth and space at the applied physics laboratory, in *Proceedings of the 1991 Rochester Forth Conference*, Rochester, NY: Forth Institute, 1991.

[Bradley, 1991] Bradley, M., private communication, 7/8/91.

[Brodie, 1981] Brodie, L., *Starting FORTH*, Englewood Cliffs, NJ: Prentice Hall, 1981.

[Brodie, 1984] Brodie, L., *Thinking FORTH*, Englewood Cliffs, NJ: Prentice Hall, 1984.

[Cox, 1987] Cox, William C., A case for NPOSs in real-time applications, *I&CS Magazine* (pub. by Chilton), Oct. 1987.

[Derick, 1982] Derick, M., and Baker, L., *The Forth Encyclopedia*. Mountain View, CA: The Mountain View Press, 1982.

[Dewar, 1970] Dewar, R., Indirect threaded code, *Communications of the ACM*, Vol. 18, No. 6, 1975.

[Dijkstra, 1970] Dijkstra, E. W., Structured programming, *Software Engineering Techniques*, Buxton, J. N., and Randell, B., eds. Brussels, Belgium, NATO Science Committee, 1969.

[Duff, 1984a] Duff, C., and Iverson, N., Forth meets Smalltalk, *Journal of Forth Application and Research*, Vol. 2, No. 1, 1984.

[Duff, 1984b] Duff, C., Neon—Extending Forth in new directions, *Proceedings of the 1984 Asilomar FORML Conference* pub. by the Forth Interest Group, P. O. Box 8231, San Jose, CA 95155, 1984.

[Dumse, 1984] Dumse, R., The R65F11 and F68K single chip FORTH computers, *Journal of Forth Application and Research*, Vol. 2, No. 1, 1984.

[Duncan, 1988] Duncan, R., (Gen'l Ed.). *The MS-DOS Encyclopedia*. Redmond, WA: Microsoft Press, 1988.

[Duncan, 1991] Duncan, R., private communication, 7/5/91.

[Electronics, 1976] RCA may offer memory-saving processor language. *Electronics*, Feb. 19, 1976, p. 26.

[FIG, 1978] *Forth Dimensions*, Vol. 1, No. 1, June/July 1978, pub. by the Forth Interest Group, P. O. Box 8231, San Jose, CA 95155.

[FIG, 1983] *Forth Dimensions*, Vol. 5, No. 3, Sept./Oct., 1983, pub. by the Forth Interest Group, P. O. Box 8231, San Jose, CA 95155.

[Gehrz, 1978] Gehrz, R. D., and Hackwell, J. A., Exploring the infrared universe from Wyoming. *Sky and Telescope* (June 1978).

[Golden, 1985] Golden, J., Moore, C. H., and Brodie, L., Fast processor chip takes its instructions directly from Forth, *Electronic Design*, Mar. 21, 1985.

[Hart, 1968] Hart, J. F. *et al.*, *Computer Approximations*. Malabar, FL: Krieger, 1968; (Second Edition), 1978.

[Hayes, 1987] Hayes, J. R., Fraeman, M. E., Williams, R. L., and Zaremba, T., A 32-bit Forth microprocessor, *Journal of Forth Application and Research*, Vol. 5, No. 1, 1987.

[Knuth, 1974] Knuth, D. E., Structured programming with **go to** statements. *Computing Reviews*, #4, 1974.

[Moore, 1958] Moore, C. H., and Lautman, D. A., Predictions for photographic tracking stations—APO Ephemeris 4 in *SAO Special Report #11*, G. F. Schilling, Ed., Cambridge, MA: Smithsonian Astrophysical Observatory, 1958.

[Moore, 1970a] Moore, C. H., and Leach, G. C., *FORTH—A Language for Interactive Computing*, Amsterdam, NY: Mohasco Industries Inc. (internal pub.) 1970.

[Moore, 1970b] Moore, C. H., *Programming a Problem-oriented Language*, Amsterdam, NY: Mohasco Industries Inc. (internal pub.) 1970.

[Moore, 1974a] Moore, C. H., FORTH: A new way to program a computer, *Astronomy & Astrophysics Supplement Series*, Vol. 15, No. 3, Jun. 1974. *Proceedings of the Symposium on Collection and Analysis of Astrophysical Data at NRAO*, Charlottesville, VA, Nov. 13–15, 1972.

[Moore, 1974b] Moore, C. H., and Rather, E. D., The FORTH program for spectral line observing on NRAO's 36 ft telescope, *Astronomy & Astrophysics Supplement Series*, Vol. 15, No. 3, June 1974, *Proceedings of the Symposium on the Collection and Analysis of Astrophysical Data*, Charlottesville, VA, Nov. 13–15, 1972.

[Moore, 1979] Moore, C. H., FORTH, The Last Ten Years and the Next Two Weeks..., Address at the first FORTH Convention, San Francisco, CA, October 1979, reprinted in *Forth Dimensions*, Vol. 1, No. 6, 1980.

[Moore, 1983] Moore, C. H., Letter to the Editor of *Forth Dimensions*, Vol. 3, No. 1, 1983.

[Paloski, 1986] Paloski, W. H., Odette, L., and Krever, A. J., Use of a Forth-based Prolog for real-time expert system, *Journal of Forth Application and Research*, Vol. 4, No. 2, 1986.

[Pountain, 1987] Pountain, R., *Object Oriented Forth*. New York: Academic Press, 1987.

[Parnas, 1971] Parnas, D. L., Information distribution aspects of design methodology, *Proceedings of IFIP 1971 Congress*. Ljubljana, Yugoslavia.

[Phys. Sci. 1975] Graphics in Kitt form, *Physical Science*, Nov. 1975, p. 10.
[Rather, 1972] Rather, E. D., and Moore, C. H., *FORTH programmer's guide*, NRAO Computer Division Internal Report #11, 1972. A later version, with J. M. Hollis added as a coauthor, was Internal Report #17, 1974.
[Rather, 1976a] Rather, E. D., and Moore, C. H., The FORTH approach to operating systems, *Proceedings of the ACM*, Oct. 1976, pp. 233–240.
[Rather, 1976b] Rather, E. D., and Moore, C. H., High-level programming for microprocessors, *Proceedings of Electro 76.*
[Rather, 1985] Rather, E. D., Fifteen programmers, 400 computers, 36,000 sensors and Forth, *Journal of Forth Application and Research*, Vol. 3, No. 2, 1985. Available from the Forth Institute, P.ÊO. Box 27686, Rochester, NY, 14627.
[Sperry, 1991] Sperry, T., An enemy of the people. *Embedded Systems Programming*, Vol. 4, No. 12, Dec. 1991.
[Taylor, 1980] Taylor, A., Alternative software making great strides. *Computerworld*, 12/?/80.
[TRS-80, 1979] Press release published in "Software and Peripherals" section of *Minicomputer News*, 8/30/79.
[Veis, 1960] Veis, G., and Moore, C. H., SAO differential orbit improvement program, *Tracking Programs and Orbit Determination Seminar Proceedings*, Pasadena, CA: Jet Propulsion Laboratories, 1960.
[Woehr, 1991] Woehr, J. J., Managing Forth projects, *Embedded Systems Programming*, May 1991.
[Wood, 1986] Wood, R. J., Developing real-time process control in space. *Journal of Forth Application and Research*, Vol. 4, No. 2, 1986.

BIBLIOGRAPHY

Brodie, L., *Starting FORTH*, Englewood Cliffs, NJ: Prentice Hall, 1981.
Brodie, L., *Thinking FORTH*, Englewood Cliffs, NJ: Prentice Hall, 1984.
Feierbach, G., and Thomas, P., *Forth Tools & Applications*, Reston, VA: Reston Computer Books, 1985.
Haydon, G. B., *All about Forth: An Annotated Glossary*, La Honda CA: Mountain View Press, 1990.
Kelly, M. G., and Spies, N., *FORTH: A Text and Reference*, Englewood Cliffs, NJ: Prentice Hall, 1986.
Knecht, K., *Introduction to Forth*, Howard Sams & Co., Indiana, 1982.
Kogge, P. M., An architectural trail to threaded code systems, *IEEE Computer*, Mar. 1982.
Koopman, P., *Stack Computers, The New Wave*, Chichester, West Sussex, England: Ellis Horwood Ltd. 1989
Martin, T., *A Bibliography of Forth References*, (Third Edition), Rochester, NY: Institute for Applied Forth Research, 1987.
McCabe, C. K., *Forth Fundamentals* (2 volumes), Oregon: Dilithium Press, 1983.
Moore, C. H., The evolution of FORTH—An unusual language, *Byte*, Aug. 1980.
Ouverson, M. (Ed)., *Dr. Dobbs Toolbook of Forth*, Redwood City, CA: M&T Press, Vol. 1, 1986; Vol. 2, 1987.
Pountain, R., *Object Oriented Forth*, New York: Academic Press, 1987.
Rather, E. D., Forth programming language, *Encyclopedia of Physical Science & Technology* (Vol. 5), New York: Academic Press, 1987.
Rather, E. D., FORTH, *Computer Programming Management*, Auerbach Publishers, Inc., 1985.
Terry, J. D., *Library of Forth Routines and Utilities*, New York: Shadow Lawn Press, 1986.
Tracy, M., and Anderson, A., *Mastering Forth* (Second Edition), New York: Brady Books, 1989.
Winfield, A., *The Complete Forth*, New York: Wiley Books, 1983.

Forth Organizations

ACM/SIGForth, Association for Computing Machinery, 1515 Broadway, New York, NY 10036. Publishers of the quarterly *SIGForth* newsletter, sponsor of annual conference (usually in March).

The Forth Institute, 70 Elmwood Dr., Rochester, NY 14611. Publishers of the *Journal of Forth Application and Research* and sponsors of the annual Rochester Forth Conference (usually in June). *Proceedings* of these Conferences are also published.

The Forth Interest Group, P. O. Box 8231, San Jose, CA 95155. Publishers of *Forth Dimensions* (newsletter), source of various books on Forth and public-domain Forth systems; also coordinator for worldwide chapters and sponsor of annual FORML conference (usually in November).

On-line Resources

BIX (ByteNet) (for information call 800-227-2983) Forth Conference. Access BIX via TymNet, then type **j Forth**. Type **FORTH** at the **:** prompt.

CompuServe (for information call 800-848-8990) Forth Forum sponsored by Creative Solutions, Inc. Type **GO FORTH** at the ! prompt.

GEnie (for information call 800-638-9636) Forth Round Table. Call GEnie local node, then type **FORTH**. Sponsored by the Forth Interest Group. Also connected via cross-postings with Internet (comp.lang.forth) and other Forth conferences.

AUTHORS' NOTE (7/95)

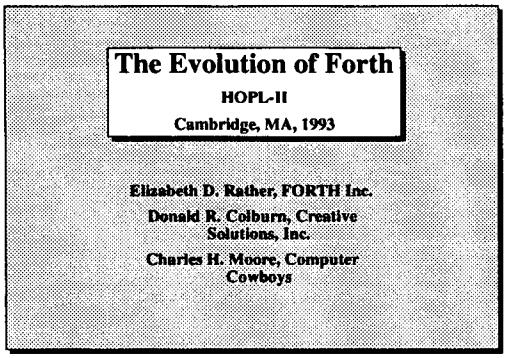
The purpose of this addendum is to present a few significant events that have occurred since its original presentation:

1. Concerning Table 13.3 (Section 13.3.2), the systems listed for Bradley Forthware are now available from FirmWorks, of Mountain View, CA, and Palo Alto Shipping is no longer in business.
2. The Forth-based underlying technology in Sun Microsystems' "Open Boot" (Section 13.3.3.1) was standardized as IEEE Std 1275-1994, *IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices*, and is available from IEEE, 345 E. 47th St., New York, NY 10017, USA.
3. ANS Forth (Section 13.5.1) received final approval March 24, 1994, and is now available as *American National Standard for Information Systems—Programming Languages—Forth*, Document X3.215-1994, from American National Standards Institute, 11 W. 42nd St., 13th Floor, New York, NY 10036.
4. The BIX and GEnie bulletin boards are now replaced by the Internet newsgroup **comp.lang.forth**.

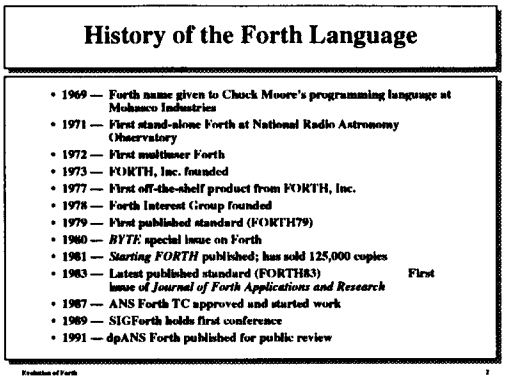
TRANSCRIPT OF PRESENTATION

SESSION CHAIR HELEN GIGLEY: Elizabeth Rather holds A.B. and M.A. degrees from the University of California, Berkeley, and an M.B.A. from Pepperdine University. Her programming experiences span FORTRAN, COBOL, BASIC, APL, NELIAC, over a dozen different assemblers, and of course, Forth. She met her first computer in 1962, the ORACLE, at Oak Ridge National Laboratory. It had vacuum tubes and reels of tape for memory. She managed data analysis for a physics group there, and went on to computing jobs in the Astronomy Department at UC, Berkeley. Later jobs included processing student records for the College of Letters and Science at Berkeley and the University of Arizona. Her first minicomputer experience was in 1971, at the National Radio Astronomy Observatory's, Tucson, Arizona, Division. Here she met Chuck Moore, the inventor of Forth, who was sitting on a high stool in front of the machine. This encounter, and the language called Forth, changed her career. Ms. Rather has been president of Forth Inc. since 1988, where she has managed the development of products and services for scientific and industrial computer applications. Currently, Forth Inc. is introducing a software development system for process control and automated manufacturing.

ELIZABETH RATHER: (SLIDE 1) I want to start out by introducing ourselves briefly. I'm sorry my two coauthors couldn't make it. They consist of Chuck Moore, who was the inventor of Forth—and with the discussion of "designed," with respect to languages and whatever, I will stick to "invention" as being the correct word in this case. He certainly never "designed" it in the sense of thinking it through. I'm not sure that if he had, he would have ever gotten started. He conceived of it, from the outset, as a personal productivity tool that, in the long run, got out of control, as some of them tend



SLIDE 1

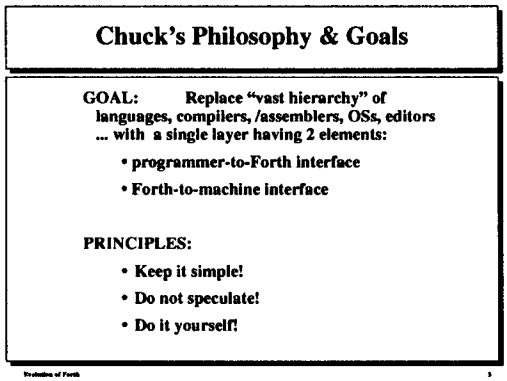


SLIDE 2

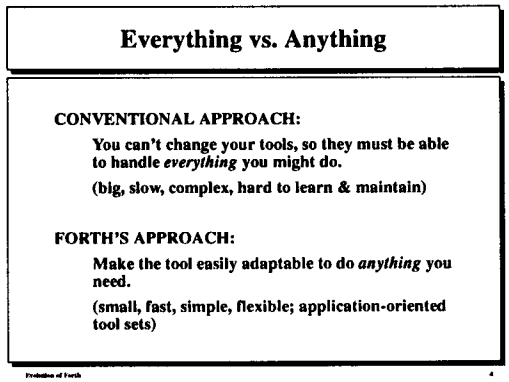
to do. For the last ten years, he has been involved in doing hardware implementations of various Forth based processors for general and specific applications, none of which have been commercially successful. Don Colburn, my other coauthor, was one of the founders of The Forth Interest Group, and one of the earliest independent developers of commercial Forth systems. He currently is the proprietor of a company that sells the most successful version of Forth for Macintoshes and also Nubus boards. He also works as a teacher's aide in an elementary school in Maryland one day a week.

(SLIDE 2) The history of Forth goes back, actually, a long way, as you can see in the paper—which I am not going to entirely summarize. Forth had precedents way back in the early '60s, in work that Chuck did at the Smithsonian Astrophysical Observatory and other places. It first acquired the name "Forth" in 1969. It was supposed to suggest a "fourth generation" computer. At the time, the "third generation" was big, and he saw the fourth generation as being distributed small computers—which in many respects was accurate. Just as "Schemer" became "Scheme," his compiler would only handle five. So the "u" went away. There has been a lot of history, much of which is covered in the paper. My talk, like others, is going to be in the form of footnotes. On the other hand, you can say that by seeing this slide, you can go home now or get ready for dinner. But, I do have a few more remarks.

(SLIDE 3) Chuck is an unusual person, and it's been a great treat knowing him. But I could not talk about Forth without talking a little bit about his philosophy and what some of his attitudes were. I have summarized it there, but just to give you a little flavor, I will read a couple of excerpts from a



SLIDE 3



SLIDE 4

book he wrote in about 1970, which was never published. He said, "In connection with these principles, do not put code in your program that might be used." He was very much opposed to the notion of "hooks"—people putting hooks in and making provisions for something that might come up in the future—he thinks that simply leads to unsanitary code. He says:

> Do not leave hooks on which you can hang extensions. The things you *might* want to do are infinite; that means that each has 0 probability of realization. If you need an extension later, you can code it later—and probably do a better job than if you did it now. And if someone else adds the extension, will he notice the hooks you left? Will you document this aspect of your program?

Well, certainly not. And even more "Chuckish" was his notion of writing everything himself. In the entire time I've known him, which is well over 20 years, he has yet to use any code that actually came from a manufacturer with their computer, including their assembler or their math (multiply and divide) routines. (In the '70s, you had to have software multiply, divide, and so on.) He said:

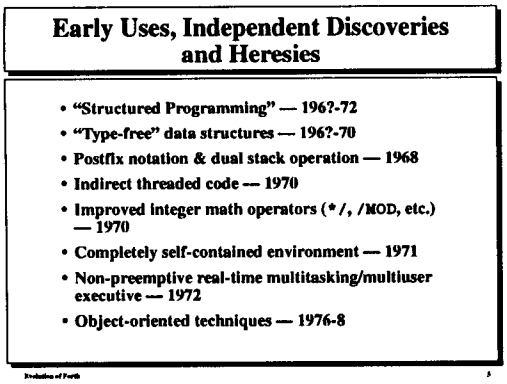
> The conventional approach, enforced to a greater or lesser extent, is that you should use a standard subroutine. I say you should write your own subroutines.
>
> Before you can write your own subroutines, you have to know how. This means, to be practical, that you've written it before; which makes it difficult to get started. But give it a try. After writing the same subroutine a dozen times on as many computers and languages you'll be pretty good at it.

(SLIDE 4) More particularly, his view was that the application programmer, who was the target of his effort, is an intelligent, responsible, creative person, who deserves and needs empowering tools. It was his perception, based on a number of years of experience, that the software tools that were provided, were generally designed with the view of the programmer as being simple minded, at best, and irresponsible in most cases (and possibly even with criminal intent), who needed externally imposed discipline, in order to stay out of trouble. He had quite the opposite approach.

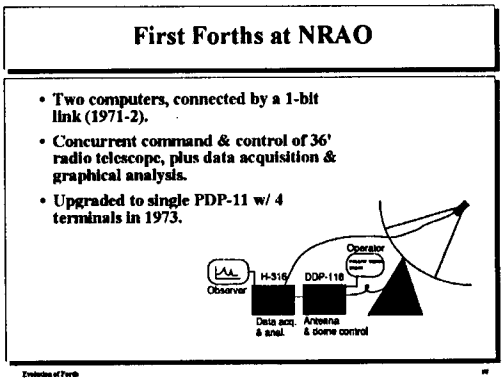
(SLIDE 5) The language that he built over the years had some unique features. This is a summary of some of them. They were unique for Chuck in that, although he spent a good part of his career around academic science, it was not computer science, and he really has never been involved in the reading and writing of papers, where these kinds of ideas are disseminated. Nonetheless, he came up with a lot of them entirely on his own. I remember distinctly when I was giving papers on Forth in the early '70s, somebody said, "What you are talking about sounds an awful lot like structured programming;" and I said, "What's that?" And they directed me to the appropriate papers, which I read with great interest and passed along to Chuck. He said, "Well I don't see what all the fuss is about; it just looks like good programming to me." And I think that is fairly representative of his ideas. He was working empirically, trying to make the kinds of tools that he saw being necessary.

(SLIDE 6) The first running, full stand-alone Forth system at NRAO (National Radio Astronomy Observatory) that I worked on, which was really the first stand-alone Forth system, was controlling a radio telescope. I found out, much to my astonishment, last Fall, that system is still in use. Unfortunately, they are about to tear down the telescope. So for all I know, the program is going to outlive the telescope—which is kind of unusual.

(SLIDE 7) Relevant to Chuck's remark about writing the same program a number of times, he actually implemented Forth personally on over 17 computers. I tried to research this list as best I could; there may be some that aren't on it. But he wrote the entire system for this many computers, and other people have written many more since. In doing one of these implementations, you had to start off by designing the assembler and writing an assembler—and then coding about a hundred assembly language primitives using that assembler, then getting it all to work. Forth was written in Forth. You used a friendly Forth computer to generate Forth for the new computer that you were



SLIDE 5

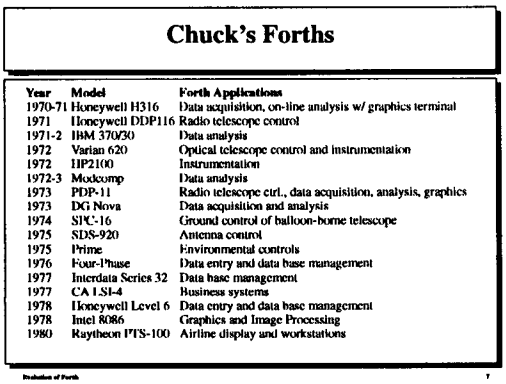


SLIDE 6

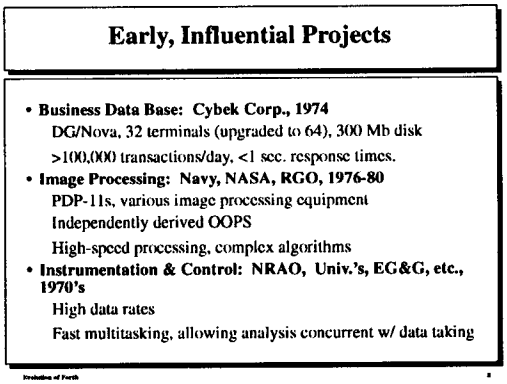
working on. But nonetheless, that's a lot of work to do. What's really astonishing is that he did this in approximately two weeks for each one—including writing the software multiply and divide routines—which were invariably faster that the ones than came from the manufacturer, which is a great source of pride to him. I would also like to point out that on several of these computers: the Honeywell 316, the (Honeywell) DDP 116, the Varian 620, the Honeywell Level 6, the Intel 8086, and the Raytheon PTS-100—which is a pretty obscure computer—on all of these, Forth was the first high-level language running on that processor. We were working on the 8086, while it was still in prototype, and in fact, helped Intel find a couple of bugs in the part.

(SLIDE 8) This is a list of some of the early projects so that you will see what kinds of applications influenced the early growth of Forth. Unlike Icon, we very definitely had applications in mind. Forth has, from day one, been designed for use in applications, and the applications in which it was used in the early days had a lot of influence. So it's been very well integrated with the real world. We traveled a lot to do these systems. Chuck built a personal portable computer in the mid-70s out of one of the first LSI-11s, which he packaged in a suitcase, and put a floppy disk in another suitcase. We carried that with us to go and install the systems on these various computers.

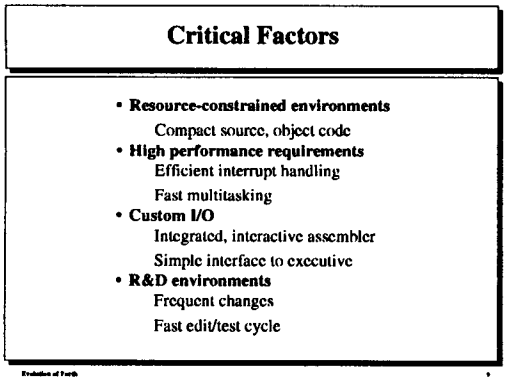
(SLIDE 9) As a result of those applications, and the influences of those kinds of applications, the critical factors in Forth became: operating in a resource-constrained environment; placing a high degree of emphasis on size and compactness of code; operating with very high performance



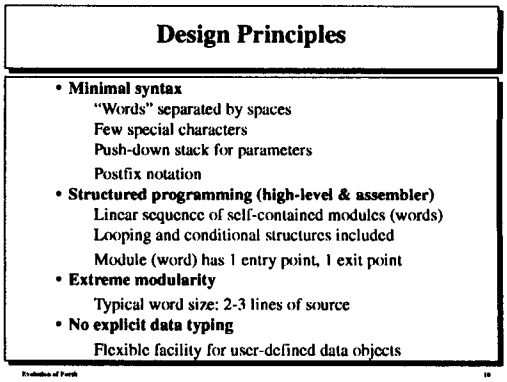
SLIDE 7



SLIDE 8



SLIDE 9



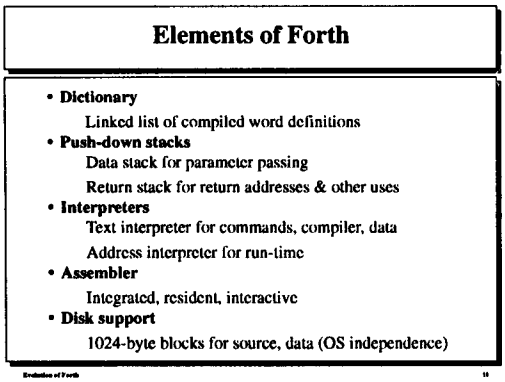
SLIDE 10

requirements, as the kinds of applications we were working on were very, very time critical; a lot of specialized custom I/O: it's very easy to write and add I/O drivers to a Forth system; and being an R&D environment, it was really important to be able to change things quickly.

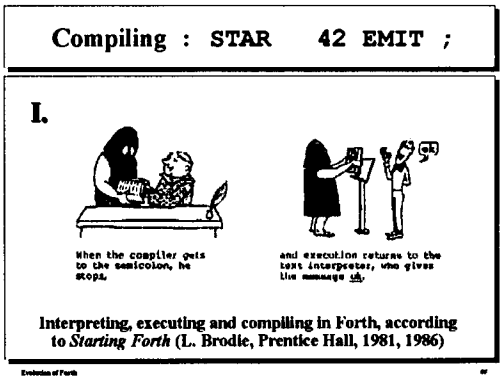
(SLIDE 10) The internal design principles that arose out of that involved a very simple language with minimal syntax. In fact, very early on when I was working on Forth, Jean Sammet wrote a book on programming languages. I wrote to her and said, "You ought to know about Forth," and she wrote back and said, "Well it's not a language; it doesn't have syntax". It, in fact, *doesn't* have very much syntax. It is, nonetheless, quite useful. Certainly, now that we found out that it does structured programming, we can say that is what it does. It is extraordinarily modular, and out of that modularity there is an effect in that programming size is far from linear. You have words calling other words. You have, in a large application, perhaps thousands of them. A word is sort of like a routine; it's sort of like a command in a language; in fact, we blur the distinctions between the two—but you develop a very, very rich vocabulary of application-oriented words in Forth. They are organized in a sort of pyramiding structure, where at the very high level you have a huge amount of leverage—by writing a line or two of code, or using just a few words, you accomplish, in fact, quite a great deal. You can do very high level operations. And—perhaps the most controversial—there is no explicit data typing in Forth at all. That flies very much against a lot of conventional wisdom, I realize. And it is very astounding to people that meet Forth for the first time. But people that have used Forth extensively find that it is one of the most valued features of the languages. While working on the ANSI Standard Report for Forth, our Technical Committee had a lot of input from a lot of sources—people wanting all their favorite word-sets or whatever. We had very, very little pressure—almost none at all—to do anything about type-checking.

(SLIDE 11) I'm not going to try to teach you the language. I will mention briefly some of the principal elements of it. One of the more unusual features is that it does include an integrated assembler in it, so that you can drop down to the assembler level at any time. It doesn't look like the manufacturer's assembler, usually, but it does produce real code.

The next three slides are from the book, *Starting Forth*, probably one of the most influential books in Forth, written by Leo Brodie, published in the early '80s, and it sold about 120,000 copies. It was very popular, but in addition to being a light-hearted book, it also includes quite a great deal of information—such as an explanation of the relationship between the compiler, the interpreter, and execution in Forth.



SLIDE 11



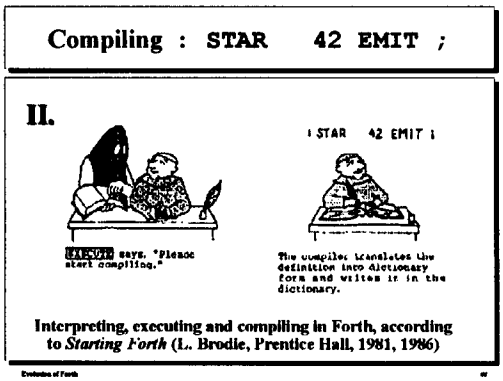
SLIDE 12

(SLIDE 12) This is our first occasion to see a Forth definition. A definition begins with colon and ends with semicolon. The word following the colon is the name of the new word being defined, and this is what it is going to do. 42 goes on a push-down stack and becomes the argument to EMIT which is going to send it to the terminal. This is how these things are put together.

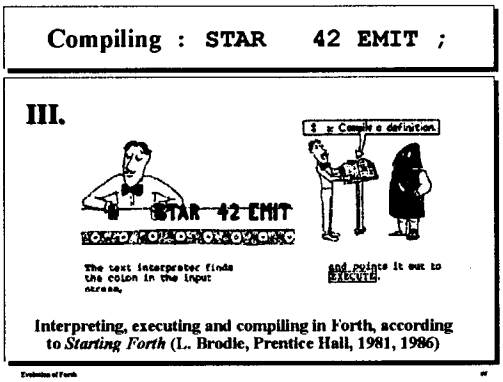
(SLIDE 13) The compiler actually executes the word : (colon), which creates the definition. It goes on until it's terminated by a ; (semicolon).

(SLIDE 14) Finally, that's done, and execution returns to the interpreter which then tells you "OK," at the terminal. In fact, that word is now compiled and is available for execution immediately. This is an interactive system that supports incremental compilation and it makes it very easy to test programs.

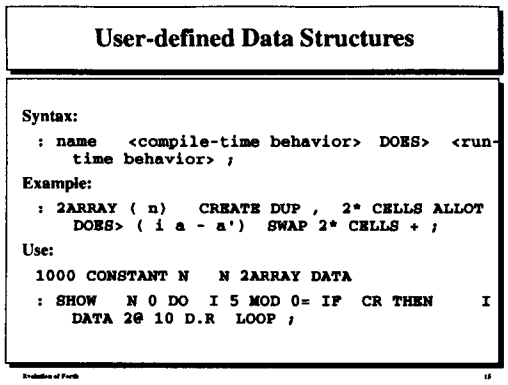
(SLIDE 15) This is a look at one of the more unusual language features, which is the ability in Forth to make custom data-types of a sort. Such definitions have two parts: There is a *compile-time* behavior and a *run-time* behavior—which will be shared by all instances of a class of words that this is defining. Here is an example: we are making an array of pairs of cells and the size of the array comes in on the stack. We make the definition, make a copy of the size, and compile it so that you can use it if you want to for range checking at run-time (although I don't think I did in this example). Then, you multiply that size by two, because there are going to be two cells for each thing, and you allot that much space. That's all you do at compile-time. Now at run-time, when a member of this class executes, it begins executing with an index on the stack that's supplied externally, and the address



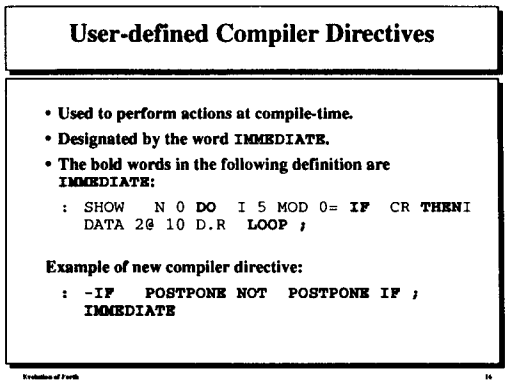
SLIDE 13



SLIDE 14



SLIDE 15



SLIDE 16

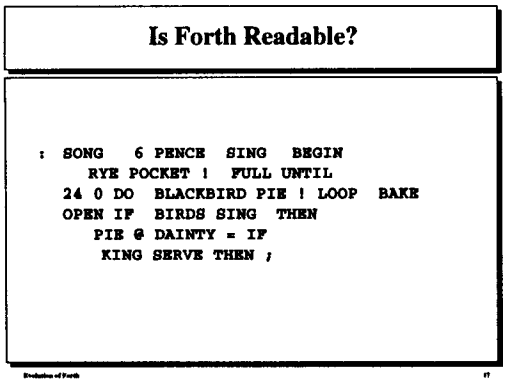
at the beginning of the array that is supplied internally, and returns the address of the item. And it does it by multiplying the index by two, converting that to cells, and adding it the address. So these things are very simple and it shows how that's used in the definition. But some of these can be very complex. You can make datatypes that live as bits on an I/O interface or very elaborate things that are application dependent. It's very easy to define such structures.

(SLIDE 16) Another interesting feature is that it's easy to add compiler directives; compiler directives are structure words. And here, for example, we have DO and LOOP, and IF and THEN. If you wanted to make a negative IF, you could do that very simply by defining POSTPONE NOT POSTPONE IF. In these two cases, POSTPONE makes a definition which is going to compile a reference to NOT and a reference to IF. When the word containing NOT IF is executed, which is executed at compile-time to create the structure. So it is very easy for a programmer to add extensions, even to the compiler.

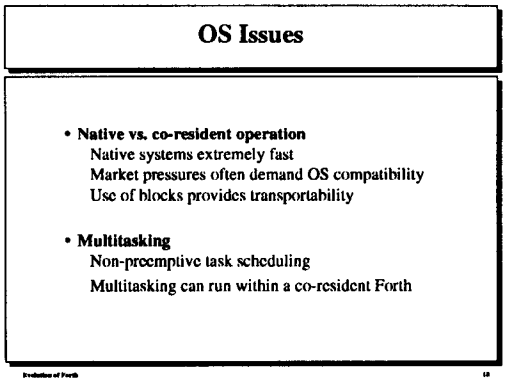
(SLIDE 17) Here is a somewhat more complex example. This shows a number of the different logical structures in it. It also has a number of application-dependent words in it. Without telling you all those words are, I think you can probably get the sense of the definition. But we have here, an indefinite BEGIN UNTIL loop; this will work until FULL returns true. The next loop is going to run from 0 to 24; it's a DO LOOP that begins *here*, and ends *there*. This is a !("Store") operator; it's going to store the value BLACKBIRD in the location PIE, and then do BAKE. Here is a conditional—when it is opened, then BIRDS WILL SING. After PIE is a @ ("fetch") operation. You fetch a value, compare it against the value DAINTY; if the result of the comparison is true, then you execute KING and SERVE, and so on.

(SLIDE 18) Many Forths have operating system capabilities. All of the Forths that I have personally ever worked on have been multi-tasking, multi-user systems, although not all Forths are that way. Many Forths are completely native, that is, they run with no host operating system. Many others run concurrently with another operating system, and there are versions available right now for most processors and most of the popular operating systems.

(SLIDE 19) These are some of the recent contemporary uses of Forth. The big areas where it is useful are, not surprisingly, those for which it was developed, that is, embedded systems and high performance control systems. Every Federal Express Courier carries one in his hand, when he picks up his package. Every Sun workstation has a Forth system on its mother-board. The "Open Boot" is currently the subject of an IEEE standard (1275) that's in development, and a number of VME system people are picking that up. There have been a lot of space applications; the control systems, just as



SLIDE 17

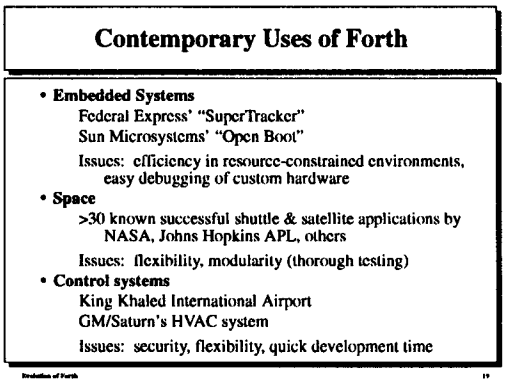


SLIDE 18

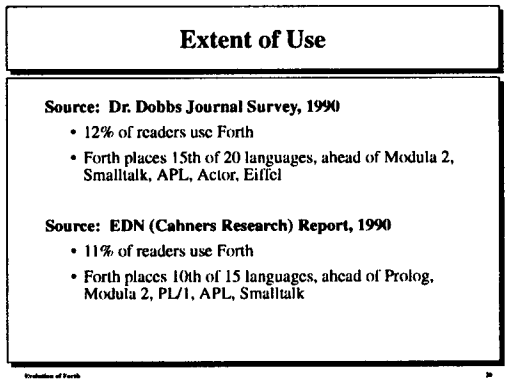
an example, the King Khaled International Airport in Riyadh, Saudi Arabia, is a system that involves approximately 500 computers. That project was done actually here in the Boston area, in the mid 1980s. There was a program that was about 300,000 lines of FORTRAN and Assembly language, that did not work fast enough or well enough. That program was replaced with an all Forth system. The 300,000 line program was, in fact, replaced by a Forth system which was written from scratch, tested, and installed in about an 18-month period, and consisted of about 30,000 lines of code. The leverage I was speaking of earlier results in considerable compactness of code, even to do a very complex application. It supports rapid prototyping very easily and works out very well in those kinds of applications.

(SLIDE 20) Quantitatively, it is very hard to measure how many people are using Forth. Our best estimates are perhaps a few tens of thousands of people. But, it is very difficult to track them all. There have been two surveys. These two magazines do surveys every even numbered year. So, I assume they did it in 1992, but I haven't seen the results yet. I just wanted to give you some measure of where it is, and the answer is, it is a minority, but it is hanging in there.

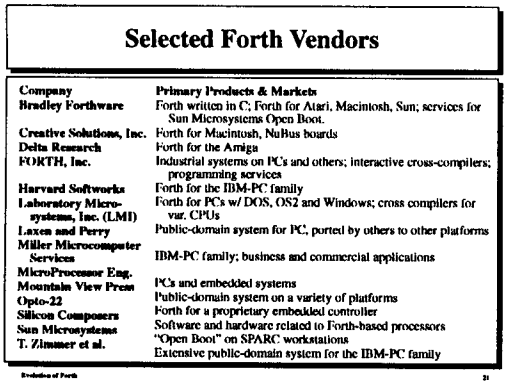
(SLIDE 21) The purpose of listing some suppliers of Forth systems is really, as much as anything, to give you a feeling for some of the diversity of some of the implementations available. There is a very definite family tree here. I was admiring the work in the Lisp paper—which I heard about—sketching out a family tree. I would love to do this for Forth. There are several major sources



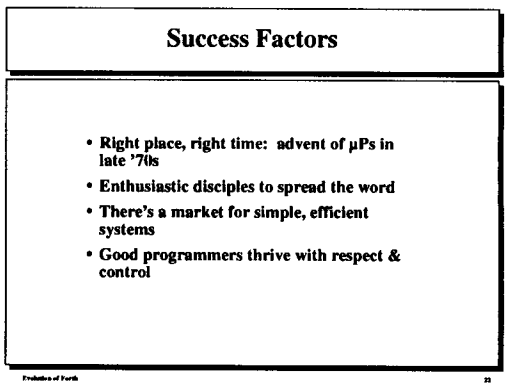
SLIDE 19



SLIDE 20



SLIDE 21

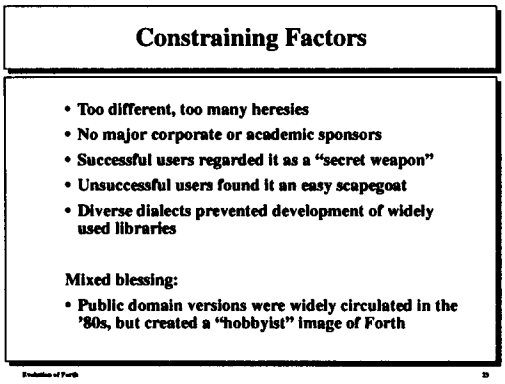


SLIDE 22

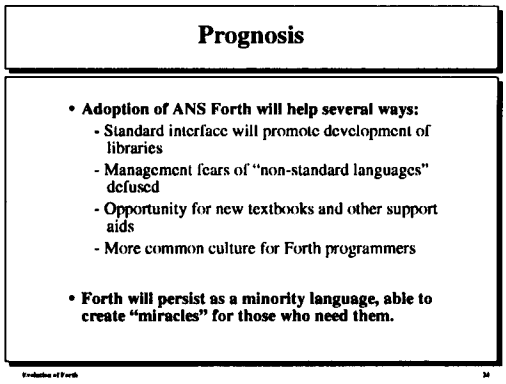
of things, ranging from some of the early commercial implementations to some of the public domain systems. There has been actually a considerable "war" in the Forth community between the vendors and suppliers of public domain systems. Remember, the early work on Forth was at a government laboratory so, therefore, the concept is public domain, but there have been a number of commercial applications. The fans of the public domain systems believe that it is in fact immoral to make money off it. Those of us who do make money off it, think that tends to contribute to language improvement and growth over time. So, it's sort of a "communist" versus "capitalist" issue that can be debated at length.

(SLIDE 22) Looking at a few success factors. Why has Forth survived now for over 20 years? It came along, in terms of its real promulgation, at the right time. In the late '70s, it was beginning to mature—and that was a time when microprocessors were there and available. They were more or less mother-naked as far as software was considered, and a lot of people became interested. A lot of very enthusiastic Forth users have spread the word, in spite of the fact that there has never been any major deep-pocket corporate or academic sponsorship at all—worse luck. And, I think, as much as anything else, the fact that it survived, means that there is a market for this kind of thing. And to look at what kind of market that is, I think it's fair to ask what kind of problem it's trying to solve. In the Ada talk, there was the issue of 2,500 line programs versus, say, 25 million line programs. And as I think I've said, these things in the world of Forth are not linear. Something that might be a 300,000 line FORTRAN and assembler program translates into 30,000 lines of Forth. It's very nonlinear there, and Forth does tend to shrink the problem. Nonetheless, it is probably true that for very, very large programs, Forth is not the best approach. I think there is the question of whether your programming philosophy, or the philosophy of your shop, really, is the theory that "a million monkeys with word processors can produce Shakespeare"—and that's going to lead to one kind of view of what software tools you need—versus the "Marine Corps" philosophy—"we need a few good programmers"—kind of thing. And that's really the philosophy that Forth is attempting to enable and support.

(SLIDE 23) So, why hasn't it taken over the world? Well, it's very different. You can write the language that looks a lot like all the other languages, and people say, "Oh yeah, I understand that." And Forth is really quite different. There are a lot of heresies, such as the lack of data typing—regardless of how well they work in practice. People are sometimes uncomfortable looking at it from the outside; it's hard to visualize how well it can work. As I said, there have never been any major corporate or academic sponsors of it, although it's used, you know we conduct guerilla warfare whenever we can; it's used in virtually all the major corporations and educational institutions. It's hanging in there!



SLIDE 23



SLIDE 24

But a lot of our users have, in fact, been reluctant to give away to the competition the secret to their success—why they can do so much more with so much less. There have also been a few projects that have not worked out, and it's been awfully easy in that case, when you have a failed project that was done in a minority language, to say, "Well of course it failed, you used that funny language." There has been some amount of publicity about a small number of "Forth disasters." I doubt that there have been any languages that haven't had their share of project disasters. If you are using the language that 99 percent of people use, however, and you have a disaster, it's a lot harder to blame it on the language, isn't it? In fact, I have looked at some of the publicized, so-called Forth disasters. I know a number of people that were involved in at least one of them, and the problems there were the same problems in just about all disaster projects, having to do with management problems, design problems, things like that. Then, finally, there has been the problem of diverse dialects. There have been several industry standards. There was one in 1979 called *FORTH-79*, that was picked up by several implementers, a much better one in 1983 called *FORTH-83*. For the last six years, there has been an ANS Forth committee (at) work.

I do want to extend my condolences to the new Smalltalk [ANS] committee. People shook their heads and clucked pityingly at me when we got started [on ANS Forth], and I'm going to do the same thing for you—and you'll find out why. We do have a draft standard out for public review—for the third time, right now—and most of us on the committee feel like "this is it!" I think that's going to help.

In the early '80s, there were a lot of public domain versions that were circulated, and that was, as I say, a mixed blessing. That got a lot of people familiar with Forth who might not have otherwise been. However, the Forth they became familiar with was of very limited functionality, sometimes poorly implemented, and certainly not a well-supported system. And if people have the notion that, "When you've seen one Forth, you've seen them all," they can hardly be blamed for getting a poor impression. However, there have been quite a number of very good implementations, both commercial systems and some of the public domain systems. I think that some people should have perhaps looked a little harder. It has been a factor in the history of the language.

(SLIDE 24) Finally, the prognosis. Adoption of ANS Forth is going to help a lot, I think, making up for some of the deficiencies. I think it's a very good standard—of course, being the chair of the committee—we've had a very, very diverse membership on the committee, from all of the dialects and so on. We have made some compromises in a number of places, but I think overall the standard is very clean and very strong. It's unfortunately about 250 pages long. *FORTH-79* was about 30 pages.

FORTH-83 was about 50 pages. At 250 (pages), this is becoming hefty. But we have done one thing that I think was very successful, and that I would recommend to some of you involved in this effort: that is, we've made it a layered standard; we have a number of optional word-sets. Since Forth was originally designed for resource-constrained environments, there's still strong feeling among Forth users that it should be possible to run Forth in a very small system. And in fact, there are Forths available that run on 8-bit single-chip micro-controllers. That is, the Forth runs on it. The whole thing, the compiler, the assembler, the whole nine yards, can run on an 8-bit single-chip micro-controller, in maybe 8 or 10K. Now, this is not one of the more full functioned Forths, but all the essentials are there. The core required word set in the ANSI standard can run in that kind of environment. Yet, you can put on optional word sets, for floating point, for host OS file access, for memory allocation, whatever you want. There are a number of these things available. I think that has been a very successful concept.

If you talked to a Forth programmer and asked him why he's so . . . people have accused Forth programmers of being sort of wild-eyed fanatics and there is some measure of accuracy in that. If you ask them why, the major reason is that they feel *empowered.* Your typical Forth programmer (and I'm not one—I have been, but I'm a mere manager now) feels that they are suddenly "superman": they've been in the phone booth and they've put on the cape. And all of a sudden, they have the power of ten. I was talking, just last night, to the local Boston Forth Interest Group, and I raised this question—and they all said that they keep getting this from customers and prospective customers, clients, whatever. A lot of them are consultants in corporations trying to persuade their bosses that they ought to be able to use Forth: that they could do in a few weeks what in C is going to take them months. There is a huge wealth of anecdotal evidence, to the effect that this is true. Certainly, as a business, we (Forth, Inc.) have succeeded by doing projects in weeks or months that were projected to take months or years. That has kept us economically viable over the years. We feel very strongly it is because of the tool, that the tool itself creates a productive environment. It's easy to point to a lot of the reasons why. But, with this amount of anecdotal evidence, we nonetheless have a credibility issue. We tell these stories and people say, "We don't believe you; all programming languages are pretty much the same, aren't they?"

I would like to close by urging somebody here, who has a sufficiently objective, respectable academic background and knows how to measure these things, and how to apply the metrics, to really take a look at this question. I know that proponents of functional programming make some of the same claims—of orders of magnitude improvement in productivity. Improvements this big deserve to be looked at. They deserve to be looked at honestly by somebody to see if there is something real there. And if so, what is it? What is the factor that makes it work? I'll be happy to supply the data if somebody wants to undertake that.

TRANSCRIPT OF QUESTION AND ANSWER SESSION

HERBERT KLAEREN (University of Tubingen): The first is, are there any significant differences between Forth and PostScript, apart from PostScript printer-specific dictionaries?

ELIZABETH RATHER: It is a very similar concept, and I'm not sure whether the developers of PostScript had seen Forth before or not. One could certainly believe that they might have. It is very similar concept; a lot of differences in detail, but I know of people who are strong in Forth or PostScript can go back and forth, so to speak, quite easily.

HERBERT KLAEREN (University of Tubingen): Can you remember how you came to base your language on the stack paradigm? Were you aware of stack-based code generation methods?

RATHER: I'm not sure. Certainly, Chuck had encountered stack somewhere along the line. I think what is really interesting about Forth, is not that it uses stacks, but in the way in which it uses stacks. The dual stack architecture has proved to be successful, probably not for any formal logical reason, but sort of the same reason the two party system in the United States works better than the 400 party system in some countries. It's a very useful number. People have experimented with different numbers of stacks and the dual stack architecture has persisted, using one stack for parameter passing and another stack for everything else—principally return information, but also a great deal else. I think, really, it's the way that it's used, as much as anything else that's been powerful.

BIOGRAPHY OF ELIZABETH RATHER

Ms. Rather was a cofounder of FORTH, Inc. and has been President since 1980. She was previously Chief Programmer for the Tucson Division of the National Radio Astronomy Observatory (NRAO), and has programmed and managed computer systems since 1962 at the University of Arizona, the University of California, and the Oak Ridge (Tennessee) National Laboratory.

She first worked with Chuck Moore, the inventor of the Forth programming language, at NRAO in 1971. Recognizing the potential capabilities of Forth, she began a campaign of talks and papers on the language that ultimately resulted in its being adopted as a standard in 1978, by the International Astronomical Union (IAU). She has authored, or coauthored, more than a dozen books and papers on Forth.

At FORTH, Inc., Ms. Rather has managed projects in a wide variety of fields, including scientific data acquisition and analysis, image processing, database management, networking, embedded systems, and industrial controls, in addition to Forth-based software development systems for over 20 platforms.

In 1987 Ms. Rather was instrumental in organizing a Technical Committee commissioned to develop an ANSI Standard for Forth. She was elected Chair of the TC, which in 1993 submitted a completed Standard (X3.215/1994) for final processing by ANSI.

Ms. Rather holds BA and MA degrees from the University of California, Berkeley, and an MBA from Pepperdine University.

BIOGRAPHY OF DONALD R. COLBURN

Don Colburn has been writing Forth Operating Systems since the late 1970s. He is the author of Multi-Forth™ and MacForth™, two popular Forth implementations for 68000 based computers. He has been active in the drafting of all Forth Standards-1979, 1983 and recent ANSI effort.

Don is a well-known developer for the Apple Macintosh family of computers. He has used the MacForth tools to write drivers for his company, Creative Solutions' popular hardware products, Hurdler™ and Hustler™. These peripherals add serial, parallel, and prototyping interfaces to the Macintosh.

Don is the father of two sons and enjoys volunteering at their schools—making them "well ahead of their time" in computerization. When he is not volunteering at the school, another very important organization he supports is the National Multiple Sclerosis Society. Don has MS and is an active source of encouragement for other MS patients.

BIOGRAPHY OF CHARLES H. MOORE

Chuck Moore was born Charles Havice Moore, II on September 9, 1938 in McKeesport, Pennsylvania. He grew up in Flint, Michigan and was Valedictorian of Central High School in 1956. He received a BS in Physics from MIT in 1960 and is a member of Kappa Sigma. While at MIT he learned FORTRAN and Lisp and programmed data reduction for Moonwatch satellite tracking and the Explorer-11 Gamma-ray Satellite. He studied mathematics at Stanford for several years, learned ALGOL and programmed electron-beam transport at SLAC.

After freelance programming on minicomputers, he learned COBOL and business programming and became an operating-system guru. In 1968, he invented Forth and used it at NRAO to program radio-telescopes. He, Elizabeth Rather, and Ned Conklin formed Forth, Inc. in 1973, to exploit the opportunities it provides. For 10 years he programmed applications in real-time control and database management. Today Forth, Inc. is a $3M firm selling software products and custom applications. Forth is particularly popular in China and Eastern Europe, where computer resources are still limited.

John Peers formed Novix in 1983, with funding from Sysorex. Its plan was to develop hardware implementations of Forth. With Bob Murphy of ICE, Moore designed a microprocessor with Forth primitives as its instruction set. Mostek produced a 4,000 gate array in 3μ CMOS that was an 8 Mips, 16-bit Forth engine. This NC4016 led to a modified NC6016 which was licensed to Harris Semiconductor in 1987. They marketed it as the RTX2000. Harris was granted two patents, with Moore and Murphy as inventors.

Computer Cowboys sold several hundred $400 Forth-Kits that incorporated the NC4016. In 1988, Russell Fish proposed a new microprocessor called ShBoom. Moore designed it and Oki Semiconductor produced prototypes on an 8,000 gate array in 1.2μ CMOS. This constituted a 50 Mips, 32-bit Forth engine.

With two technically successful gate-arrays, Moore wanted to produce a custom design. He was by now convinced that existing tools were inadequate. So in 1990 he started developing unique layout and simulation software. Layout is based upon five layers of square tiles that produce correct-by-design chips. A simple transistor model simulates the entire part and its connections. This was first implemented on ShBoom, and later ported to a 386.

He then designed MuP21 with a 20-bit bus and four 5-bit instructions/word as a way to minimize memory cost (only five 4-bit DRAMs). The internal buses are 21 bits to provide a 1M-word addresses for both DRAM and SRAM. MuP stands for Multi-uProcessor. The intent is to achieve parallelism with several independent microprocessors sharing memory. ShBoom had a DMA controller. MuP21 has a video generator—NTSC output with 16 colors—and a memory manager with DRAM, SRAM, and cache timing. Specialized processors can be very simple and effective.

Prototypes have been made by Orbit Technologies on their mosaic wafers, and by HP, through the MOSIS prototyping service where 12 to 25 chips cost $3,000 to $6,000 with two-month turn-around. This low cost makes possible iterative development, though project time can be long. After several tries, the design tools and architecture converged to a 100 Mips, 21-bit Forth engine. It is now being upgraded to 300 Mips.