VOLUME XI, NUMBER 6 MARCH/APRIL 1990

FORTH DIMENSIONS

DESIGN TRADEOFFS IN STACK COMPUTERS

A 32-BIT FORTH ENGINE

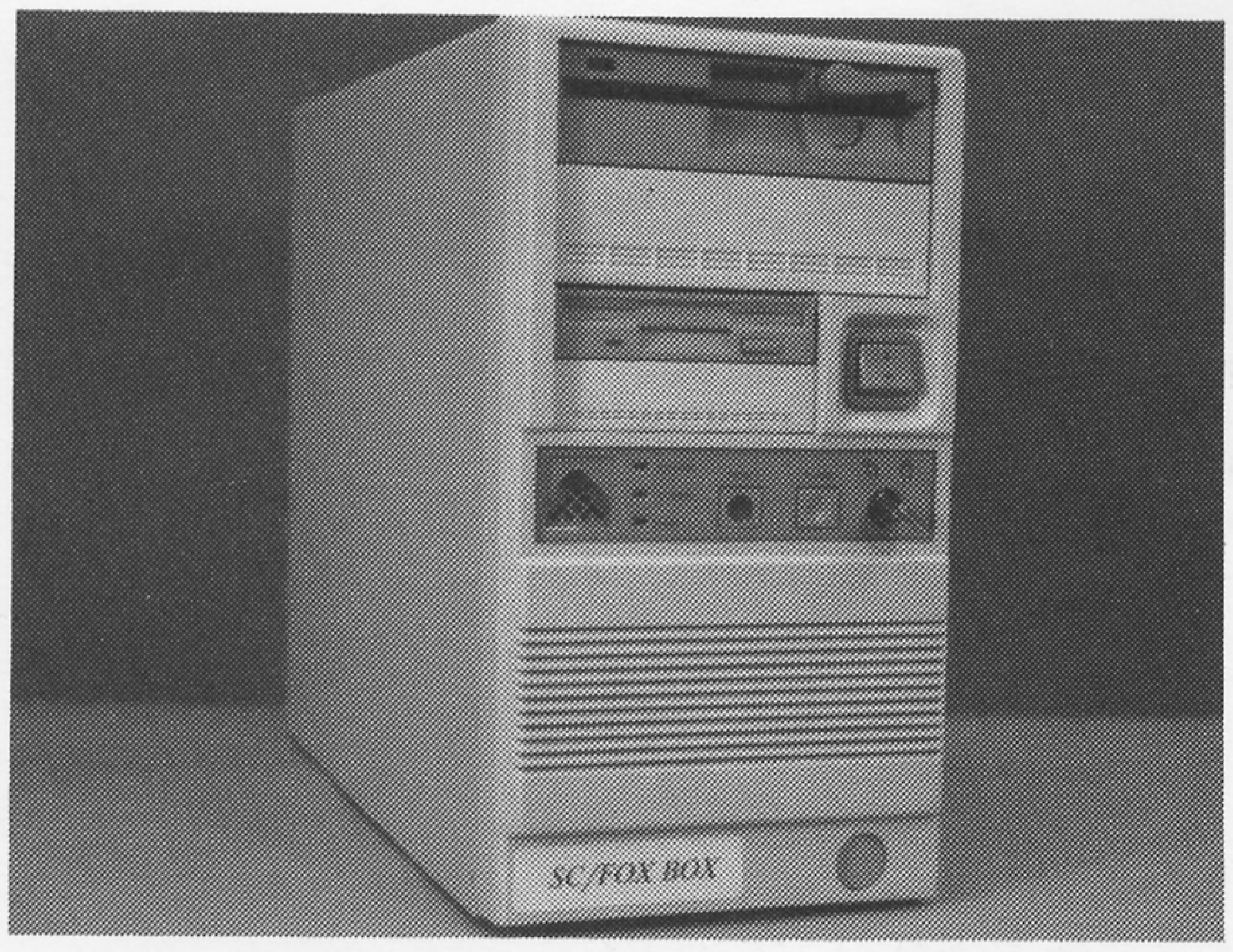
PHASE ANGLE DIFFERENCE ANALYZER

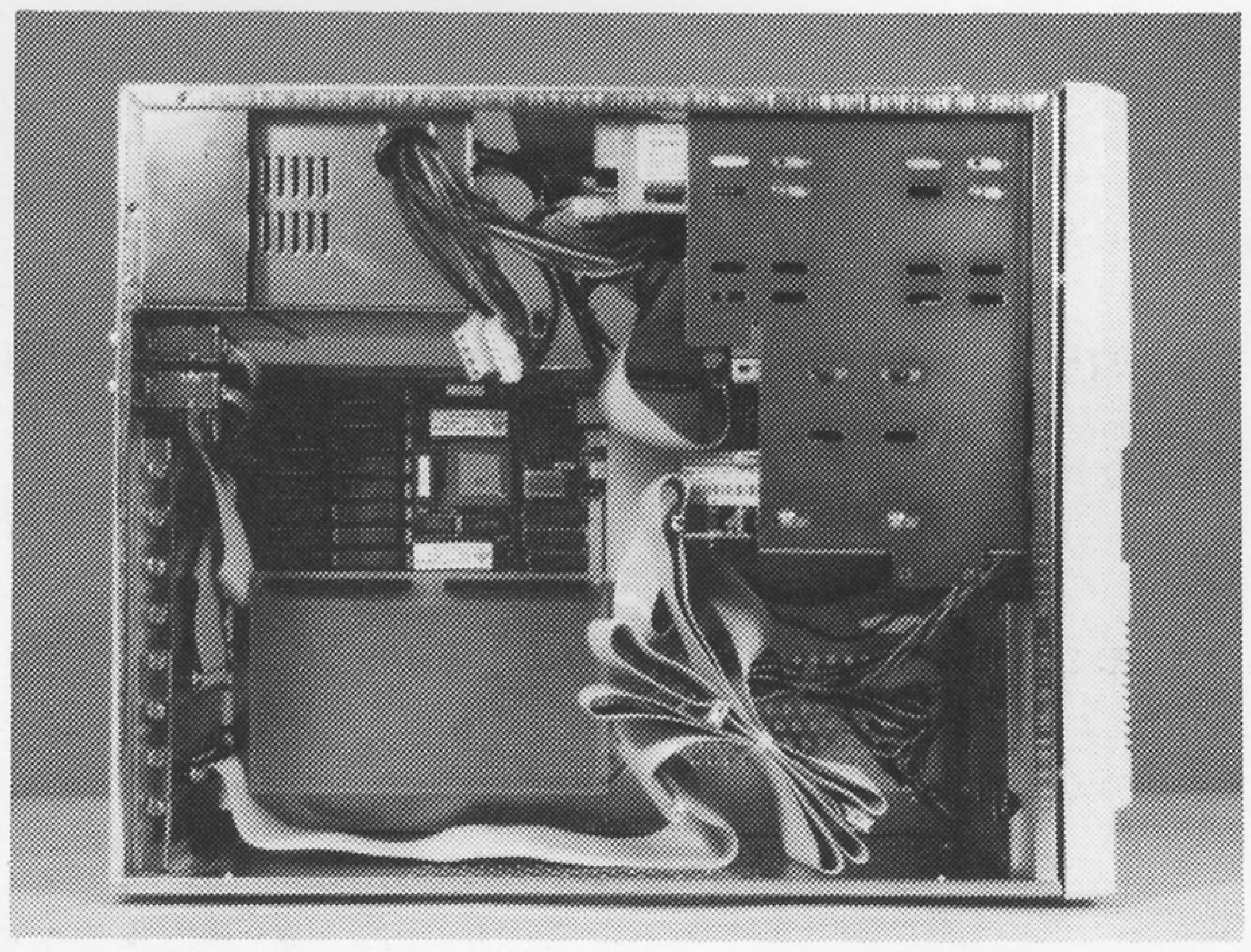
INTERPRETATION MACROS

ANS FORTH: HARDWARE INDEPENDENCE

SILICON COMPOSERS

SC/FOX™ BOX Computer System





The SC/FOX (Forth Optimized eXpress) BOX is a high-speed stand-alone computer, designed both as a real-time development platform and a system controller for embedded applications. Mounted inside is the SC/FOX Single Board Computer, a small (100x160mm) stand-alone computer that uses the 16-bit RTX 2000 Forth CPU, and the SCSI I/O board. Communication with the SC/FOX BOX takes place across a 56K-baud RS-232 serial port, to either a PC compatible computer or a stand-alone terminal. The SC/FOX BOX boots from its on-board EPROMs, and loads software or data from the SCSI hard disk, floppy drives, or serial port.

SC/Forth Language
* Interactive language, Forth-83 Standard.
* Automatic optimization and μcode support.
* Time-slice or user-defined multitasking.
* Code overlay system and program spawning.

Mini-Tower Enclosure: 13"x16"x7".
* 200-watt power supply.
* Two 5¼" half-height bays and three 3½" half-height bays (one for 3½" hard disk).
* Optional full-tower enclosure.
* Wyse 150 terminal and software.
* Custom configurations quoted.

SC/FOX SBC Single Board Computer
* Uses Harris RTX 2000™ Forth Chip.
* 8, 10, 12, or 14 MHz operation.
* 64K to 512K bytes zero-wait state SRAM.
* 56K baud serial terminal port.
* Centronic parallel printer port.

SC/FOX SCSI I/O Daughter Board
* SCSI port for disk drives, optical drives, etc.
* Floppy disk port supports up to 4 disk drives.
* 56K baud serial port.
* 16-bit bidirectional latching I/O port.

External Ports
* 9-pin RS-232 terminal and serial port.
* 25-pin Centronic parallel port.
* 40-pin bidirectional parallel port.
* 50-pin SCSI I/O port.

Standard Mini-Tower Configuration
* 8 MHz SBC with 128K bytes SRAM.
* SCSI I/O board with software drivers.
* Two 5.25" 1.2M byte floppy drives.
* One 40M-byte SCSI hard disk drive.
* SC/Forth in on-board shadow EPROM.
* Room inside for custom hardware.

Ideal for embedded real-time control, high-speed data acquisition and reduction, image or signal processing, or computation-intense applications. For additional information, please contact us at:

SILICON COMPOSERS INC 208 California Avenue, Palo Alto, CA 94306 (415) 322-8763

FORTH DIMENSIONS

EDITORIAL

Forth Hardware

This issue contains the top three articles selected from those we received in response to our call for articles about Forth hardware. It was a successful experiment, in my jaded editorial eyes, because it brought us a number of very good manuscripts (only three of which are presented in this issue—we will publish others, with their authors' permission, in upcoming issues). There was an interesting split decision in the judging, calling for editorial arbitration, but Phil Koopman's "Design Tradeoffs in Stack Computers" received a unanimous vote for first place. Second place went to John Hayes for "The SC-32: a 32-Bit Forth Engine," while third place went to Dr. C.H. Ting's "Phase Angle Difference Analyzer." Cash awards will be sent to those three authors in recognition of their contributions. We are honored and pleased to bring you their work.

Times have changed since Glen Haydon and Chuck Moore closeted themselves in Glen's computer-riddled crow's nest, densely wire-wrapped boards lying like disemboweled mazes atop the gurneys they used for workbenches. Chuck left to develop what would become the NC2000 for Novix (a device that will probably be remembered only as the first real Forth chip). Well, the hardware bug bit some of the best minds in the Forth world, and it bit them hard. Perhaps they sensed, as Jack Woehr suggested in the last issue, that Forth as we have known it all these years is—at its most metaphysical roots—an evolving description of an ultra-efficient microprocessor architecture. Or perhaps it was just that Forth's way of seducing us into hardware intimacy led us to believe we could do anything.

In any case, soon we had a selection of interesting devices to tinker with. Industrious efforts (some realized and some not) sprang out of small shops and universities, and there were Zilog's Super8 and Rockwell's R65F11. These actually became bread-and-butter hardware for some Forth programmers.

Re-enter Glen Haydon, who had teamed up with Phil Koopman, Jr. Soon the Haydons' loft was streaming out schematics, and the two of them were selling wire-wrap kits and PC boards as the promising WISC (i.e., writeable instruction set computer) CPU/16 and CPU/32. These were stack-based devices whose native instruction sets could be changed about as easily as a Forth definition, and they blazed right along at fine speed. Phil also dove into a doctoral program; his résumé must have left the entrance examiners a bit breathless, unless they are accustomed to candidates who have already implemented working examples of a promising, untried microprocessor architecture. Much of his interesting research has been published as *Stack Computers, The New Wave*.

The kicker is that the CPU/16 and /32 drew the attention of Harris Semiconductor. Harris negotiated for the rights to develop this technology, and since then have invested considerably in its success. They incorporated the WISC concepts in their standard cell library and produced the RTX 4000 microprocessor; the RTX 2000 is their Novix successor.

This string of developments, which continues to unfold, offers hope for the future employment of Forth programmers: if any large chip maker manages to pinpoint the conjunction of Forth's strengths and the market's evolving needs, there could be a decided upswing for hardware experts, systems vendors, developers, consultants, and plain-old programmers. Already,

(Continued on page 34.)

Forth Dimensions
Published by the
Forth Interest Group
Volume XI, Number 6
March/April 1990
Editor
Marlin Ouverson
Advertising Manager
Kent Safford
Design and Production
Berglund Graphics

Forth Dimensions welcomes editorial material, letters to the editor, and comments from its readers. No responsibility is assumed for accuracy of submissions.

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $30 per year ($42 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 8231, San Jose, California 95155. Administrative offices and advertising sales: 408-277-0668.

About the Forth Interest Group

The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. Many research hardware and software designs that will advance the general state of the art. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $24/36 per year by the Forth Interest Group, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 8231, San Jose, CA 95155."

DESIGN TRADEOFFS IN STACK COMPUTERS
A PERSONAL EXPERIENCE

PHILIP KOOPMAN, JR. - WEXFORD, PENNSYLVANIA

When I started designing stack processors for WISC Technologies in 1985, little had been published about the architectural requirements of Forth engines. A substantial amount of architectural measurement had been performed on previous stack-based processors (in particular the Xerox Mesa architecture), but the behavior of single-stack processors for executing conventional languages is not representative of the types of things Forth processors do. When I started, all I knew was that Forth programs did a *lot* of subroutine calls, but beyond that I was groping in the dark. Here I hope to describe some of the history behind the development of the WISC and 32-bit RTX processors in terms of discoveries, blunders, and serendipity. Along the way, I will talk about the various requirements for implementing a high-speed Forth engine, and will describe the motivations underlying the design of Harris' 32-bit RTX architecture.

THE HARDWARE-FRENZY PHASE

The first phase of my continuing journey to stack-computer enlightenment was characterized by a frenzy of designing, building, debugging, and programming Forth hardware.

The WISC CPU/16

The WISC CPU/16 was my first stack computer design (and, for that matter, my first computer design of any type). The "WISC" stands for Writable Instruction Stack Computer. It was implemented entirely in 74LSxxx series TTL components, wire-wrapped on a single IBM-PC plug-in board. We produced a printed circuit board version once the design was shaken out. The design decisions for the CPU/16 were made in favor of simple and inexpensive prototyping first and foremost. This led to the decision to use a microcoded design, with RAM chips for a writable control store instead of a hardwired design.[1] A block diagram of the CPU/16 is shown in Figure One.

The design had 256 elements for each stack, and 256 opcodes with eight possible micro-instructions per opcode. Most instructions took three micro-cycles to execute, with subroutine calls and returns tying up the data bus to the exclusion of other operations. Figure Two shows the two instruction types supported: subroutine call and opcode. Thus, the importance of Forth's subroutine call was incorporated, but the rest of the design was dictated primarily by the constraints of fitting everything onto a single board while still using standard TTL components.

The RISC vs. CISC battle was about to take a new turn...

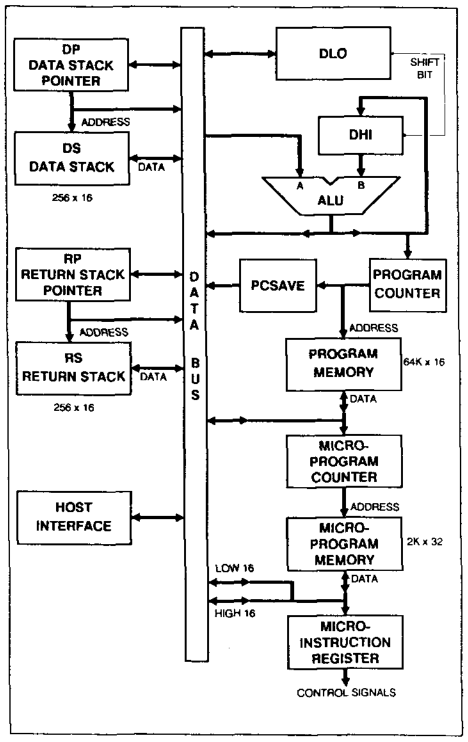
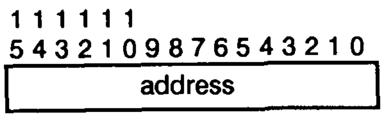


Figure One. WISC CPU/16 block diagram.

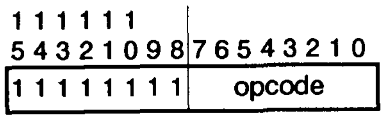
The Novix NC4000 chip had been introduced shortly before the WISC CPU/16 was built. A principle difference between the two designs (other than the fact that the Novix was a single chip compared to the CPU/16 discrete implementation) was that the Novix was a hardwired processor, while the CPU/16 was microcoded. The simplistic microcode implementation techniques used on the CPU/16 caused it to take an average of three micro-instructions for each opcode (at a cost of three clock cycles). At similar clock speeds (which translated into similar program memory speeds), one would have expected the NC4000 to outperform the CPU/16 by a factor of three to one.

But that didn't happen. Instead, the 4.77 MHz CPU/16 was much slower than a 5 MHz NC4000 on programs that used simple operations, but competitive (although, probably, not quite as fast) on programs that used more complex operations. This was because complex operations,

1. This decision was perhaps influenced by the fact that I did not possess an EPROM programmer, and that available programmable logic for use in synthesizing random logic was very modest in capabilities—and I didn't have a programmer for that either.

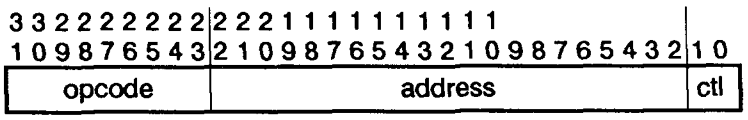


Bits	Function
0-15	Subroutine address (bits 8-15 of the address must not all be 1)



Bits	Function
8-15	All 1, specifying an operation instruction
0-7	Opcode

Figure Two. CPU/16 instruction formats.



Bits	Function
23-31	Opcode
2-22	Address for jump or call (word aligned)
0-1	Program flow control
	00 Jump / 10 Call
	01 Return / 11 unused

Figure Three. CPU/32 instruction format.

such as double-precision math and multi-element stack manipulations, were implemented in microcode in fewer clock cycles than the equivalent sequences in NC4000 assembly language. The execution speed for a mix of Forth primitives was just under one million typical Forth operations per second (including complicated operations such as multiply and double-precision math in a typical instruction mix).

As a result of my CPU/16 experience, I think microcoded techniques are inappropriate for a 16-bit Forth processor in most cases. Primarily, this is because the requirements for 32-bit wide microcode cause a single-chip implementation to be too large to be competitive with a hardwired approach. Also, the use of a microcoded approach does not provide many additional benefits when the processor is restricted to a 16-bit instruction format. However, the experience showed that something interesting was possible—microcoded machines could, perhaps, be competitive with hardwired machines with similar functions. This was because flexibility of operation and a high semantic content in each instruction could make up for a lack of raw speed. In other words, the RISC vs. CISC battle was about to take a new turn in the arena of stack computers.

The Monster/32

WISC Technologies produced a single prototype of a 32-bit computer that was seen by a very few people at the 1986 Rochester Forth Conference. In his book *The Mythical Man-Month* (Addison-Wesley, 1982), Fred Brooks describes what he calls the "second system syndrome." In this syndrome, the designer of a system saves up scores of neat ideas that can't be implemented in the first system because of time and money constraints. When the designer gets another crack at a similar problem (the second system), all these ideas are thrown in, usually with disastrous results.

The Monster/32 was my second system. The only truly good idea that was included was the decision to make it a 32-bit machine. Some of the ideas were reasonably good, but poorly executed. One idea was the inclusion of extra registers around the ALU. This eliminated congestion caused by having to save and restore the top-of-stack register when using the ALU for other calculations. Another idea was the addition of separate hardware to increment subroutine return addresses independent of the ALU.

The worst ideas had to do with the micro-instruction format and the use of multiple counters for addressing program memory. The 64-bit micro-instructions had a large number of interesting features, including the capability to specify a variable length for each micro-cycle. None of these features turned out to be very useful. The complexity of the micro-instruction format did result in almost impenetrable microcode that was very difficult to write and debug.

The Monster/32 was constructed using eight wire-wrapped boards in an S-100 card cage (but without using the S-100 bus in the usual manner). The wire-wrapping exercise itself taught me an important lesson about the value of simplicity, and wore out my first electric wire-wrap gun.[2] The system was eventually operational for a period of two weeks, and successfully ran a Forth system. The folks who saw it operate at the Rochester Forth conference never did ask why the attachment cable to the IBM PC host was only a foot long. There was an incredible noise problem in the host interface, and any longer cable wouldn't work reliably.

It became clear that, for a number of reasons, my first 32-bit design was a flop. Fred Brooks, again in *The Mythical Man-Month*, asserts that you should always be prepared to "throw one away." So we did.

2. Based on this experience, I rate battery-powered wire-wrap guns at about two miles of wire per gun.

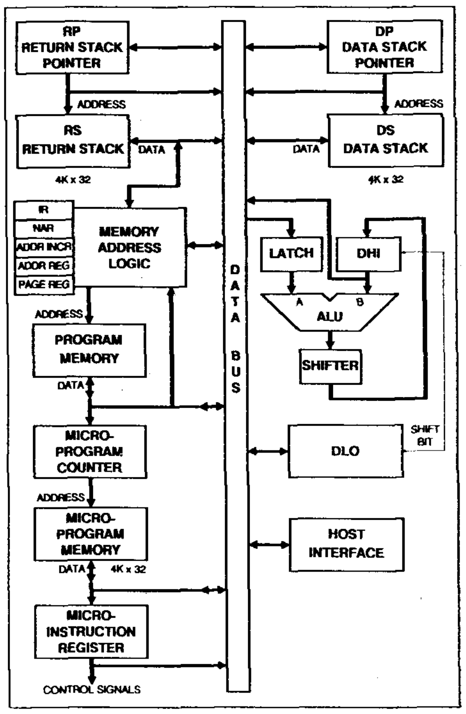


Figure Four. WISC CPU/32 block diagram.

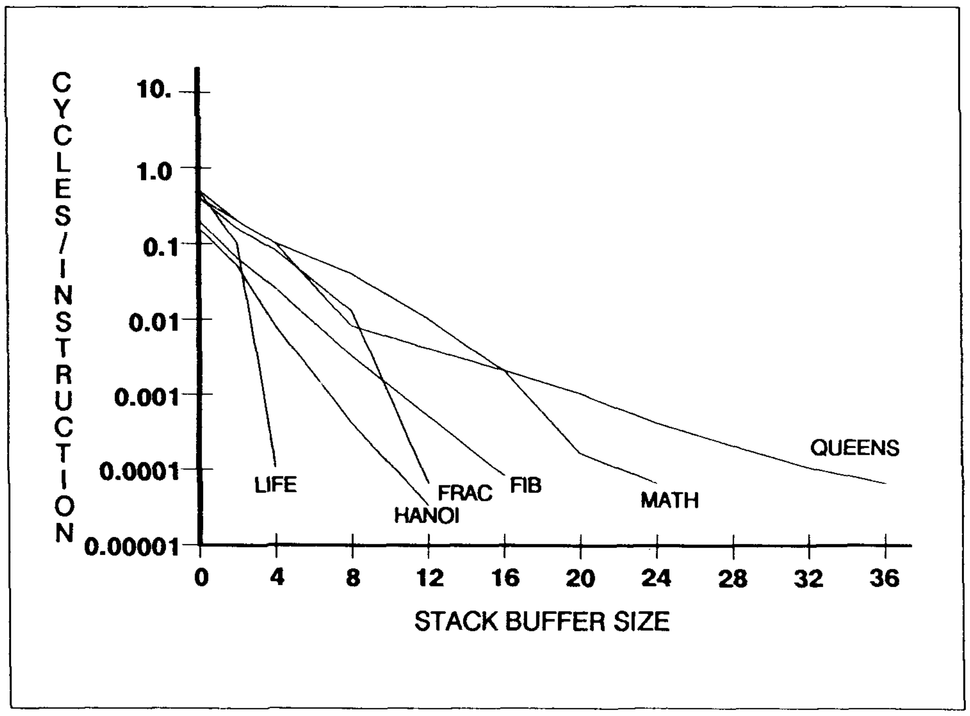


Figure Five. Return stack spilling overhead vs. stack buffer size.

The CPU/32

I began to distill the Monster/32 experience, and to decide what formed the true essence of an efficient WISC system. The CPU/16 had been arbitrarily constrained to simplicity, whereas the Monster/32 had been allowed to grow almost limitlessly. While there were a few good ideas to be salvaged, overall my immensely complex 32-bit design was a waste of good silicon. I began to see what I had missed in the realm of hardware design, despite my extensive experience with Forth: within limits, simpler *is* better.

At the same time, I began to combine several ideas that had been collecting in the back of my mind. One of them was that CPU cycle times can be made much faster than affordable memory speeds. Another was that taking advantage of concurrency in operations is a traditional way of speeding up computers that I had not exploited very well in previous designs. The last major idea was that, since microcoded stack machines only need eight or nine bits to specify an opcode, much of my 32-bit instruction memory was being wasted as unused bits in opcode-type instructions.

The answer to all my collected concerns hit like a bolt of lightning one day.[3] There were enough bits left over in an opcode instruction to also hold a large address, so why not make *every* instruction have both an opcode and a subroutine call? This had the effect of reducing program size, as well as providing for simultaneous operation of subroutine calls and opcodes. Thus, the resulting machine allowed control flow (subroutine calls and returns) to proceed in parallel with data manipulations (data stack operations), allowing two separate operations to be accomplished on each instruction. In other words, it offered the ideal situation for a Forth programmer: subroutine calls for free. Of course, in order to have a complete set of machine operations, a subroutine return format was required, which also combined an opcode with the return operation.

Not every instruction was a subroutine call or return, so there was a need for an instruction that incremented the program counter as well. In my quest to simplify the hardware, I made another discovery: the program counter was unnecessary. By using a jump instruction format instead of an increment-PC instruction format, I could have every instruction point to the next instruction to be executed (even if it was just the next sequential instruction). This reused the logic that performs subroutine calls, with a modification to suppress the push of the return address onto the return stack. The instruction format of the CPU/32 is shown in Figure Three.

Other enhancements to the CPU/32, based on experiences with the Monster/32 and the limitations of the CPU/16, included using a latch between the bus and the ALU to facilitate single-cycle exchange of data between the DHI register and the Data Stack. The microcode format was trimmed back to 32 bits, which makes microcode simple enough to be easily comprehensible, and saves a large amount of memory space. A block diagram of the CPU/32 is shown in Figure Four.

Another important insight in the design of the CPU/32 was the balance achieved between program memory speed and processor speed. RISC processors strive to execute one instruction per clock cycle. That implies that memory must be cycled as quickly as the clock in order to provide a steady stream of instructions. In a simple and streamlined processor, that means that programs must reside in fast memory. Usually, the required memory chips are so expensive that even high-end RISC systems must use them sparingly as cache memories. Many Forth applications have traditionally been in the areas of real-time

3. I don't remember just how the idea came to me. My best ideas usually come during my morning shower. However I was not electrocuted, so this one probably did not.

control. Many real-time control applications cannot afford the unpredictability of cache memory. Many others can't afford the cost of even a single bank of fast memory chips for any purpose. So, taking advantage of the fact that a microcoded machine can have a higher instruction semantic content (*i.e.*, it can accomplish more work per instruction), I designed the CPU/32 to execute an instruction every *two* micro-cycles, with each memory bus cycle taking two clock cycles. Assuming that both micro-cycles of every instruction are well employed, this allows twice the processing power for a given memory speed than an approach of one instruction per clock cycle.

The CPU/32 was originally built on reused S-100 boards from the Monster/32, with 74ALSxxx logic and some 74Fxxx logic for speed-critical sections. The use of "F" logic caused enough noise problems that the wire-wrapped version never ran at speed, so we produced a printed circuit board version before debugging was completed. This five-board version eventually ran at a 6 MHz micro-cycle rate, and executed approximately three million Forth operations per second.

The RTX 32P

The finished CPU/32 was demonstrated at the 1987 Rochester Forth Conference. At that conference, Harris Semiconductor was promoting its RTX 2000 processor, a redesign of the NC4000. They were intrigued by the possibilities for the CPU/32 as a 32-bit member of the RTX family. So, in July of 1987, I visited Melbourne Florida and transferred the schematics of the CPU/32 into their standard cell design system. In 31 days, the design was entered and verified with the help of one Harris engineer.[4] The product of this effort was, in January of 1988, an implementation that was functionally identical to the three printed circuit boards of the CPU/32 core processor, reduced to two chips operating at an 8.3 MHz micro-cycle rate. The two chips were the data chip (with the ALU, data stack, and half the microcode memory) and the control chip (with the memory addressing logic, the return stack, and the other half of the microcode memory).

The reason for a two-chip set instead of a single-chip processor implementation was to allow maximum flexibility with the finished system. 2K words of microcode memory were included on-chip, since 256 opcodes seemed to be more than I could possibly use.[5] When asked how big the stacks should be, I replied, "Gee, how much will you give me?" So, the chips ended up with 512 elements by 32 bits each for data and return stacks. This resulted in three things: it allowed Harris to make the biggest chip they have ever attempted, it made for a poor yield, and it produced chips which have logic on one quarter and memory in the other three. But, all these results were in keeping with the experimental nature of the project.

THE ANALYSIS PHASE

After the successful production of the CPU/32, I began to define and build a commercial version of the architecture for inclusion in the RTX product family. This exercise involved optimizing the architecture to fit the design constraints of CMOS chip technology as well as evolving the architecture to improve performance and better address the needs of the marketplace.

In the summer of 1987, I foolishly agreed to simultaneously refine the architecture for Harris and write a book about stack computer architecture. I did survive the summer, and found that the synergy between the two tasks was amazing. The book required me to think about measuring and describing the essence of stack machines. The design task required me to think about efficiency and architectural refinement. By the end of the summer, I had reached a number of conclusions about tradeoffs in stack machine design.

Stack Size

One of the big unknowns in producing the RTX 32P was how big to make the stacks. Before, I had been limited either by the need to keep chip count low or by standard high-speed memory chip sizes. On the RTX 32P, I guessed at 512 stack elements.

I guessed wrong. Simulations of several Forth programs show that many programs never used more than four or five stack elements. Of those that used more stack elements, all showed a small variability in stack size across reasonably large periods of time. In order to reduce hardware costs, it is advantageous to exploit this behavior and reduce on-chip stack sizes to the minimum possible.

An interesting line of thought to pursue is to assume that on-chip stacks are so expensive that they will be smaller than required. Also assume that there is some mechanism (say, a finite state machine that monitors stack overflows and underflows) that will copy elements to and from memory as required. The question to ask, then, is how much does this copying cost in terms of program performance degradation? Figure Five shows the results of a simulation for the return stack on a number of programs. The vertical axis indicates the amortized costs of stack spills in terms of wasted memory cycles per instruction executed in the course of the program. Notice that this axis has a logarithmic scale. The horizontal axis specifies the size of the on-chip stack buffer. The amazing thing is that, for a stack size of 16 elements, the cost is less than one percent. For a stack size of 16 to 32 elements, the cost reduces to essentially zero. Data stack behavior is similar.

The right answer, then, to how big stacks should be is 16 or 32 elements, no more. In the case of a multitasking environment, it is advantageous to have a partitioned stack that allocates 16 or 32 stack elements for each task in order to eliminate context-switching overhead.

Hardwired vs. Microcoded Performance

With the design of the RTX 32P, the hardwired control vs. microcoded control issue became ripe for detailed study. The RTX 2000 and the RTX 32P represent two processors designed to accomplish similar tasks using similar technology. One is hardwired, the other microcoded. The question is, which is faster?

I collected statistics on instruction execution frequency for Forth programs. But, I didn't simply gather numbers for the obvious primitives such as `DUP`, +, and `SWAP`. Instead, I took an IBM PC Forth compiler that was optimized to the point that anything worth speeding up was written in assembly language. This became my set of Forth "primitives"; that is, the basic building blocks used by real Forth code in real programs. Not surprisingly, these primitives included many double-precision operations (including "2-type" stack opera-

4. That includes the weekend I took off to visit Walt Disney World.

5. Of course this means that they were completely filled with mostly worthless junk almost immediately.

tions), and slow instructions such as multiply and divide. After I had measured the instruction execution frequencies for several programs, I multiplied the frequency times the number of clock cycles required for each of the RTX 2000 and RTX 32P processors. I assumed that RTX 2000 programs were operating on 16-bit data, and that RTX 32P programs were operating on 32-bit data. The result was surprising.

Despite the fact that most instructions on the RTX 2000 execute in a single clock cycle and that all instructions on the RTX 32P execute in two or more clock cycles, the RTX 32P required only ten percent more clock cycles than the RTX 2000 to do the same amount of work. In other words, clock-for-clock, the two processors did about the same amount of work. Part of the reason for the RTX 32P's good performance was the fact that its microcoded opcodes mapped well onto the high-level Forth operations used in real programs. Another part of the reason was that many of the subroutine calls counted as instructions, but were executed "for free" by the RTX 32P when combined with opcodes in the same machine instruction. Note that, although the program execution speed is similar, the RTX 32P accomplishes the same amount of work in half the memory accesses as the RTX 2000, since it accesses memory every two clock cycles. This difference allows it to use much slower memory for comparable processing speeds.

The result of this comparison is that it is not clear that the RISC approach of hardwired instructions and single-clock-cycle execution offers a compelling benefit over microcoded designs in terms of program execution speed for stack machines. This means that designing a 32-bit processor with hardwired control may result in suboptimal use of available memory bandwidth. For reasons previously stated, this should not be interpreted as meaning that 16-bit Forth chips should be designed with microcoded control—the area costs are just too high, and the lack of bits in the instruction format to support simultaneous opcode and subroutine call execution makes the potential payoff too low.

C—The Realities of the Marketplace

Forth is Good. But, Forth doesn't always Sell. The fact is, C is becoming the language of choice in many application areas, including real-time control. Also, architectural features required to support C go a long way towards supporting Ada for the military market. So, the RTX family is migrating to a position in which C is the primary language for many users. Forth then becomes the "assembly language" for the system, used for optimizing critical routines.

Aside from minor quirks of C (such as signed and unsigned characters, requiring optional signed byte extension on byte fetches), the only important C structure that is incompatible with Forth-based stack machines is the stack frame. C semantics *assume* that anything in the stack frame is addressable as a normal memory element. Furthermore, C stack frames grow too big to fit into any reasonably sized on-chip stack buffer. So, a stack processor must have some efficient method of supporting a stack frame. At a minimum, this means having a dedicated frame pointer on-chip, as well as the capability for using frame-pointer-plus-offset addressing. The RTX 2000 design incorporated a movable User Area pointer that can fulfill this requirement (an improvement over the NC4000, which had a fixed User Area location). The RTX 32P did not have this capability, but you can be assured that the commercial 32-bit RTX chip will.

For Forth users, the frame pointer can provide unexpected benefits. Many Forth programmers have advocated the use of local variables of some sort as a way of improving code organization and readability. A frame pointer mechanism makes an ideal implementation vehicle for a local variable stack, as well as providing a clean interface between C procedures and Forth subroutines.

Conclusions

I've described some of the history behind the sequence of processors leading up to the 32-bit RTX chip now in development. Along the way, I've tried to give some insight into why the processors have been designed the way they have, and into stack machine design issues in general. While the information has been presented as a personal history, it should provide some idea of the essential elements of designing stack computers.

In the real world, design of a good architecture is seldom done entirely through the sole use of wisdom and knowledge, and is never done right on the first try. Happenstance, and the background and education of the designer have much to do with the process. More important than the ability to get it right the first time is the ability to recognize mistakes, try new ideas, and retain the best of the old while incorporating the best of the new.

I would like to take this opportunity to acknowledge the involvement of two people without whom this history could not have taken place. Glen Haydon provided insight, encouragement, and financial support for the WISC Technologies processors. Dave Williams has been personally responsible for the acceptance and survival of the RTX 32-bit technology at Harris Semiconductor.

Philip Koopman Jr. is a senior scientist at Harris Semiconductor .

ADVERTISERS INDEX

Forth Interest Group	44
Harvard Softworks	16
Inner Access	18
Institute for Applied Forth Research	38
Journal of Forth Application & Research	36, 37
Laboratory Microsystems	18
Miller Microcomputer Services	24
Next Generation Systems	14
Silicon Composers	2, 26

SC32: A 32-BIT FORTH ENGINE

JOHN HAYES - LAUREL, MARYLAND

Over a period of several years, a group of us at the Johns Hopkins University Applied Physics Laboratory have designed a series of microprocessors that directly execute the Forth programming language. The SC32 is the third in the series.

Interpreted languages have a big advantage over compiled languages, in their shorter software development cycles and their ability to interactively test and debug programs. This is especially true with Forth, whose simple syntax allows it to be extended in application-unique ways. Unfortunately, the performance of Forth on conventional computers suffers when compared to compiled languages because of the need for run-time interpretation. Forth-oriented processor chips, by treating Forth as object code, eliminate run-time interpretation. Consequently, interactive Forth programs running on the SC32 execute just as fast as equivalent compiled programs on conventional microprocessors.

All of the SC32's internal and external data paths are 32 bits wide. The chip has an external 32-bit address bus and 32-bit data bus. 16-bit processors will be useful in many applications for years to come. However, once the size of a program or its data exceeds the memory addressable by a 16-bit processor, the bank switching and segmentation schemes that must be resorted to hurt system performance. The flat, linear address space of a 32-bit processor provides the most convenient programming model for large applications.

Forth Direct Execution

The SC32 directly executes the Forth programming language. From this description, you might think that the chip reads in Forth source code and executes it. However, the reality is less exotic and more subtle. Three aspects of the SC32 make it a Forth direct-execution engine: elimination of run-time interpretation, an instruction set optimized for Forth, and an internal processor data path designed to support Forth stack-based programming. Each of these points is discussed in the sections that follow.

Most Forth primitives are implemented with one instruction.

Eliminating the Inner Interpreter

Forth is implemented on traditional processors using the virtual machine approach shown in Figure One. Because of the mismatch between Forth's stack model and the native processor, a layer of run-time interpretation is necessary. A tiny assembly language program called the inner (or address) interpreter is written for the bare processor. Forth's primitive stack operators, also written in assembly language, are implemented in the kernel layer. The top layer of a Forth system, the interactive outer interpreter is written in Forth. The SC32 processor implements the inner interpreter and kernel layers in hardware, eliminating run-time interpretation. This means that Forth programs running on the SC32 execute as fast as equivalent compiled programs while retaining Forth's interactive environment.

The inner interpreter in traditional Forth systems uses a technique called threaded code [Rit80]. Figure Two shows how the Forth program:

```
: mod  /mod drop ;
```

is compiled on an indirect-threaded code system. /MOD has been previously defined using : (colon). In the body of MOD's definition, /MOD and DROP are represented as pointers (threads) to their respective definitions. DROP is a primitive and its definition is in assembly language. The definition of /MOD, another colon definition, consists of threads to its constituents.

When MOD is executed, the inner interpreter traces through the list of threads, nesting down when necessary, until a primitive word defined in assembly language is found. Control is then transferred to the primitive. In Forth systems implemented on traditional processors, 35–50% of the system's time is consumed by the inner interpreter.

The SC32 eliminates this run-time overhead by eliminating the inner interpreter. Figure Three shows the MOD example compiled for the SC32. Instead of a pointer to DROP, the actual object code for DROP appears within MOD's definition. The pointer to /MOD is replaced with a subroutine call to /MOD. At run time, a list of SC32 instructions is traced, instead of a list of pointers. The inner interpreter has become the fetch-execute cycle of the processor.

Readers familiar with advanced Forth implementation techniques will realize that the scheme described above is subroutine threaded with in-line code expansion. Theoretically, nothing precludes using this technique on conventional processors. However, the mismatch between Forth and typical instruction sets would cause compiled Forth programs to become much larger. For example, if several instructions are needed to implement DUP on a given

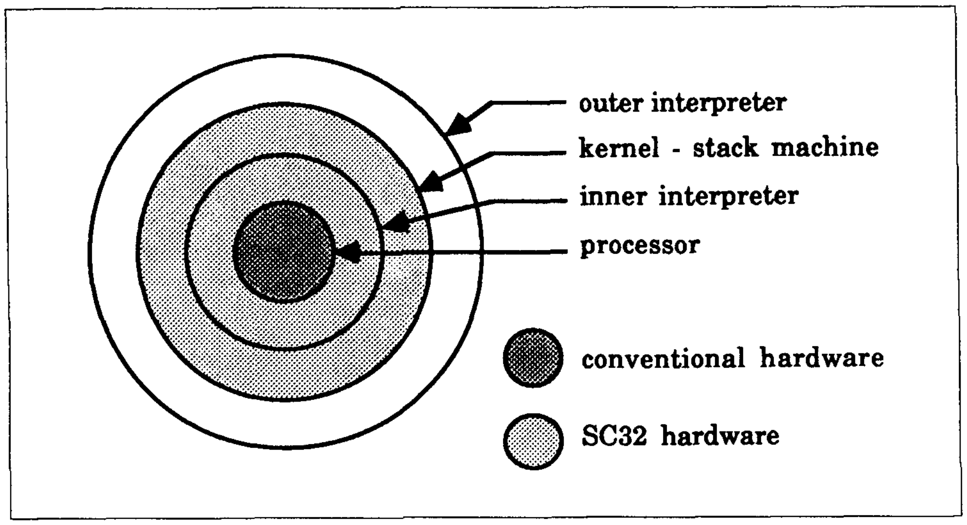


Figure One. The Forth virtual machine.

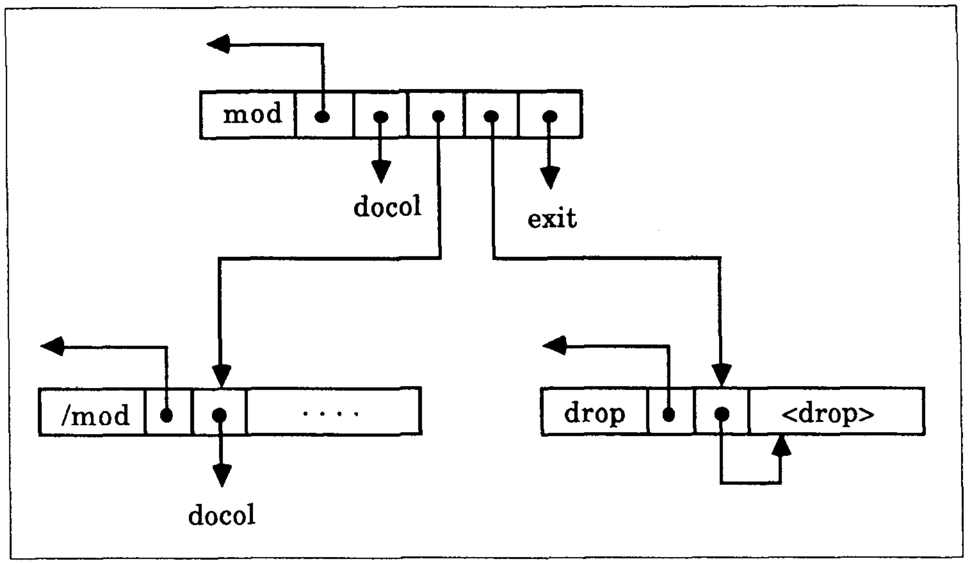


Figure Two. Indirect-threaded code.

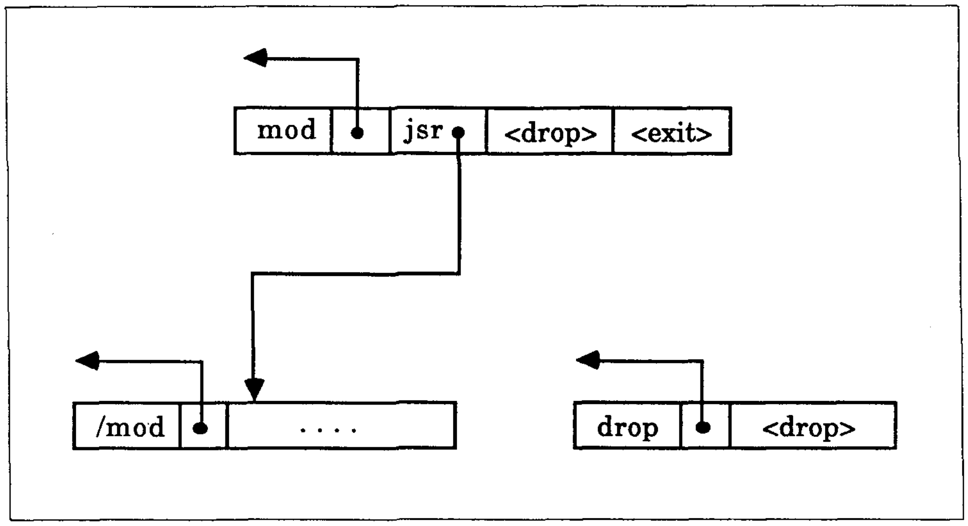


Figure Three. Direct execution.

processor, the resulting object code could be significantly bigger than the size of a thread. The instruction set of the SC32 has been designed so that almost all Forth primitives are implemented with one instruction. This is the subject of the next section.

Instruction Set

The SC32 instruction set was designed specifically to support the Forth programming language. There is a one-to-one mapping between most Forth primitives (DUP, +, DROP, etc.) and SC32 instructions. All of the instructions execute in one machine clock cycle, with the exception of load-from-memory and store-to-memory, which take two cycles. Consequently, a complete Forth primitive is executed almost every clock cycle.

All SC32 instructions are 32 bits wide. Having 32 bits to represent instructions allows a chip designer to create a more regular instruction set with less instruction decoding circuitry needed on chip. Less decoding logic allows the chip's clock rate to be increased, reducing the time needed to execute an instruction. The instructions come in three categories: control flow, loads/stores, and arithmetic operations. Table One shows each instruction type available within a given category. There are a total of eight instruction types, with the three most significant bits of the instruction determining its type.

The notation used to describe the instructions in Table One is unusual. Traditional assembly languages, consisting of an operation followed by operands, are inadequate for describing the SC32's instruction set. Instead, a register transfer notation is used in Table One and throughout this article.

There are three control flow instructions: subroutine call, branch, and conditional branch. These instructions all contain an embedded destination address. Measurements show that a single-cycle subroutine call is the most important ingredient of a Forth engine. Calling a colon-defined word from within another colon word is the most frequently executed operation in Forth programs. The bit-level instruction encoding of the call instruction was chosen so that the processor interprets any 32-bit pointer (thread) into the low words of memory as a subroutine call.

Fast branch instructions are important too. A single-bit condition code flag determines whether or not the conditional branch is taken. The flag must be set by an earlier instruction.

The load/store category of instructions consists of load-from-memory, store-to-memory, load-address-low, and load-address-high. In these instructions (and in the micro-instruction described below), *R1* and *R2* are operand selectors. Possible operands include the top four locations on Forth's parameter stack, the top four return stack locations, and miscellaneous registers. The load-from-memory instruction takes the operand specified by *R1*, adds a 16-bit *Offset* value to form a memory address, fetches a value from memory, and places it where indicated by *R2*. Store-to-memory is similar, but the *R2* operand is stored at the computed memory address. These two instructions take two clock cycles to execute. In the load-address-low instruction, the address computation is performed as described above. However, instead of fetching from memory, the address is put in *R2*. The load-address-high is similar, but the *Offset* value is shifted left sixteen bits before being added to *R1*. The load address instructions execute in one clock.

The micro-instruction (so called because of its similarity to conventional microcode) performs arithmetic, logic, and shift functions. One ALU operand is selected by *R1* and the other operand is always on the top of the parameter stack. The ALU result is stored in *R2*. Much detail about the micro-instruction has been suppressed in Table One. The *ALUop* control field actually has two formats, one for controlling shift operations and the other for controlling arithmetic operations. The arithmetic format has six subfields for selecting the arithmetic operation, the source of the carry input, and the result to be loaded in the condition code flag. The shift format is similar, except that the arithmetic operation subfield is replaced by four shift control subfields.

Micro and load/store category instructions also have the ability to control Forth's stacks and select the source of the next instruction using the *Stack* and *Next* fields. *Stack* can specify that any combination of pushing and/or popping the parameter and return stacks should occur in parallel with the execution of the instruction. *Next* selects either the program counter (PC) or the top of the return stack (TOR) as the source address of the next instruction. Usually, the PC provides the address. However, if TOR is specified in *Next* and the return stack is popped, a return-from-subroutine is done concurrently with the execution of the current instruction. In other words, a subroutine return occurs in zero time. (Note that the similar Novix return bit always popped the return stack [Gol85].)

There are a number of programming tricks that are useful on the SC32 and that help in the implementation of Forth. The single addressing mode, register indirect plus offset, is more powerful than it appears. This simple addressing mode subsumes the functions of several other modes. For example, a register indirect mode results from setting the offset to zero. The SC32 has an internal register that always returns the value zero when read. This allows the construction of an absolute addressing mode by adding the offset to the zero register. The offset becomes an absolute address. More complex addressing modes can be constructed using more than one instruction.

The load address instructions also provide some tricks. Programmers familiar with conventional instruction sets might have been surprised by the absence of a move instruction in Table One. Adding a zero offset to the *R1* operand in a load address instruction is equivalent to moving *R1* into *R2*. A move immediate instruction that loads a literal value into *R2* can be produced by setting *R1* to the zero register. The result is that the offset is loaded into *R2*. Any 16-bit literal can be produced in one clock cycle using this trick. Any 32-bit literal can be constructed in two clocks using a load-address-high followed by a load-address-low.

Data Path

The data path of a processor is the organization and connectivity of internal resources such as registers and ALUs. The data path, along with some control logic, implements the processor's instruction set. Several elements of the SC32 data path (such as the zero register and condition code flag) have already been mentioned. However, the most important features of the data path are two stack caches, one for the parameter stack and one for the return stack. The stack caches are key to executing one Forth primitive every cycle. The SC32 stack caching algorithm guarantees that the top four values of both stacks are always present in the chip. Consequently, Forth primitives always find their operands on chip and never need extra memory cycles to fetch them.

A stack cache is implemented as a sixteen-word circular buffer. As Forth primitives push or pop words from the stack, the words are added to or removed from the buffer. When the buffer fills, hardware intervenes and inserts two cycles to write a word from the buffer into external memory. When the buffer is almost empty, hardware again intervenes to read a word from external memory back into the buffer. To use the stack caches, the programmer loads a register with the address of the external overflow region. Subsequently, the operation of the stack cache is completely transparent and gives the programmer the illusion of arbitrarily large on-chip stacks.

The concept of a stack cache will probably be new to most readers. Once the idea sinks in, you might wonder how well it performs. Nothing is free, and the extra cycles added on buffer overflow and underflow will slow down a running program. However, measurements show that this "slow down" is less than one percent for typical Forth programs [Hay88]. The depth of the stack oscillates around an average value for long periods of time. The cache attempts to adjust itself so that the buffer is centered on the average depth and captures as large a range of depth variations as possible.

Forth on the SC32

Now that we've examined the three elements of the SC32 direct execution engine, it's time to see how Forth is implemented on the processor. Table Two shows some representative Forth primitives implemented with the SC32 instruction set. The stack operations that push or pop the parameter stack are denoted by ↓P and ↑P.

DUP is implemented with a load-address-low instruction. The value on the top of the parameter stack (TOS) is read into the ALU and zero is added to it. In the meantime, the parameter stack is pushed, allocating a new top-of-stack slot. Then the ALU result is written into the slot. The entire operation takes one clock cycle. Notice that in Table Two, operand selectors to the right of the arrow refer to the state of the stack after the push or pop has occurred. Thus, the two uses of TOS in DUP refer to two different storage locations.

A slew of Forth data movement primitives can be built with the load-address-low

control flow	Type:3	Address:29				
	call					
	branch					
	conditional branch					

load/store	Type:3	Next:1	R1:4	R2:4	Stack:4	Offset:16
	load:			$*(R_1 + \text{Offset}) \rightarrow R_2$		
	store:			$*(R_1 + \text{Offset}) \leftarrow R_2$		
	load address low (lal):			$R_1 + \text{Offset} \rightarrow R_2$		
	load address high (lah):			$R_1 + \text{Offset} \cdot 2^{16} \rightarrow R_2$		

micro	Type:3	Next:1	R1:4	R2:4	Stack:4	ALUop:16
	micro:			R_1 ALUop TOS $\rightarrow R_2$		

Table One. SC32 instruction set.

Primitive	SC32 Instruction
`DUP`	TOS → TOS; ↓P
`>R`	TOS → TOR; ↑P; ↓R
`R>`	TOR → TOS; ↑R; ↓P
`OVER`	SOS → TOS; ↓P
`1+`	TOS + 1 → TOS
`1234`	ZERO + 1234 → TOS; ↓P
`12345678`	ZERO + 1234 · 2^{16} → TOS; ↓P
	TOS + 5678 → TOS
`@`	*(TOS + 0) → TOS
`!`	*(TOS + 0) ← TOS; ↑P
	↑P
`+`	SOS + TOS → TOS; ↑P
`<`	SOS — TOS; $N_{xor}V$ → FL → TOS; ↑P
`0=`	TOS; Z → FL → TOS

Table Two. SC32 implementation of some typical Forth primitives.

Sequence	SC32 Instruction
`OVER 1+`	SOS + 1 → TOS; ↓P
`R> +`	TOR + TOS → TOS; ↑R
`+ 0=`	SOS + TOS; Z → FL → TOS; ↑P
`AVARIABLE @`	*(ZERO + AVARIABLE) → TOS; ↓P
`OVER @`	*(SOS + 0) → TOS; ↓P
`OVER ANARRAY + @`	*(SOS + ANARRAY) → TOS; ↓P
`DUP 9 + @`	*(TOS + 9) → TOS; ↓P
`BEGIN`	
`  DUP SIZE < WHILE`	ZERO + 8190 → TOS; ↓P
	SOS — TOS; $N_{xor}V$ → FL; ↑P
	conditional branch <forward>
`  0 OVER FLAGS + !`	*(TOS + FLAGS) ← ZERO
`  OVER +`	SOS + TOS → TOS
`REPEAT`	branch <back>

Table Three. SC32 object-code compaction.

instruction. `DUP`, `>R`, `R>`, and `OVER` in Table Two are examples. Primitives that add small (16-bit) constants to TOS (e.g. `1+`, `2+`, etc.) are implemented with load-address-low. Small literals are created with load-address-low by adding the literal value to the zero register and pushing the result on the parameter stack. Table Two also shows how a large literal is constructed by pushing the most significant part of the number on the stack, then adding in the least significant part. Forth's `@` and `!` operators are implemented with the load-from-memory and store-to-memory instructions, in the obvious way.

Forth has several arithmetic primitives that operate on the top two parameter stack elements, pop the stack, and write the result to the top of the stack. Examples of such binary operators are `+`, `AND`, `XOR`, etc. These operators are implemented with the micro-instruction. *R1* selects the second item on the parameter stack (SOS) as one ALU operand (remember that the other operand is always TOS in micro-instructions). The ALU performs the selected arithmetic or logic operation, the stack is popped, and the result written to TOS. Binary comparison primitives such as `<`, `=`, `U<`, etc. are also implemented with the micro-instruction. Micro has the ability to produce a 32-bit 0 or -1 truth value as the result of a comparison. Unary arithmetic primitives (e.g., `NEGATE`, `1-`, `NOT`, etc.) and unary comparisons (`0<`, `0=`, `0>`, etc.) can all be realized with one micro-instruction.

By now, the reader should have a good feel for how Forth's primitives are implemented on the SC32. However, this is not the whole story. The SC32 instruction set and data path are more general than the pure stack model needed for Forth. As a result, it is possible to map multiple Forth primitives into a single SC32 instruction. Consider the sequence of primitives `OVER 1+`. `OVER` works by sending SOS through the ALU unmodified and pushing the value onto the stack. `1+` now reads this value through the ALU again, this time adding one. The first movement through the ALU is superfluous and can be eliminated by combining `OVER 1+` into one instruction.

Table Three has several more compaction examples. Each entry in the table suggests an entire class of compactions. There is an astronomical number of possible combinations. An optimizer written for the SC32* captures most of the more useful cases. All of the examples in the table, including the last one, were compacted by the optimizer without human intervention. The last example, the inner loop of the Sieve of Eratosthenes [Gil83], was included to test your understanding of the SC32.

Wrapup

The SC32 microprocessor is fabricated in a two µm CMOS process and packaged in an 84-pin PGA. We are using the SC32 on a number of projects within JHU/APL, the most interesting being the flight computer of a magnetometer processor that is part of a Swedish satellite named Freja. The chip is also available commercially from Silicon Composers in Palo Alto, California.

Our Forth chip-design team consists of Martin E. Fraeman, myself, Susan C. Lee, Robert L. Williams, and Thomas Zaremba. A description of the SC32's predecessors can be found in [Hay87a] or [Hay87b]. A more complete description of the SC32 will appear in [Hay89].

References

Gilbreath, J., Gilbreath, G. "Eratosthenes Revisited: Once More through the Sieve," January, 193, pp. 283–326.

Golden, J., Moore, C.H., Brodie, L. "Fast processor chip takes its instructions directly from Forth," *Electronic Design*, March 21, 1985, pp. 127–138.

Hayes, J.R. "An Interpreter and Object Code Optimizer for a 32-Bit Forth Chip," *1986 FORML Conference Proceedings*, pp. 211–221.

Hayes, J.R., Fraeman, M.E., Williams, R.L., Zaremba, T. "An Architecture for the Direct Execution of the Forth Programming Language," *Proceedings of the Second Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 42–49.

Hayes, J.R., Fraeman, M.E., Williams, R.L., Zaremba, T. "A 32-Bit Forth Microprocessor," *Proceedings of the 1987 Rochester Forth Conference*, pp. 39–48.

Hayes, J.R., Lee, S.C. "Stack Caching in the SC32 Forth Processor," *1988 FORML Conference Proceedings*, pp. 100–104.

Hayes, J.R., Lee, S.C. "The Architecture of the SC32 Forth Engine," *Journal of Forth Application and Research* 5,4 (to appear).

Ritter, T., Walker, G. "Varieties of Threaded Code for Language Implementation," *BYTE* 5,9, September, 1980, pp. 206–227.

John R. Hayes received an M.S. in computer science from Johns Hopkins University in 1986. He has written flight software in Forth for satellite-based magnetometer experiments and for the shuttle-based Hopkins Ultraviolet Telescope. He is currently applying the SC32 in a variety of projects.

*[Hay86] describes an optimizer written for our first Forth chip. While its instruction set differs from that of the SC32, the optimization techniques are similar.

NGS FORTH

A FAST FORTH, OPTIMIZED FOR THE IBM PERSONAL COMPUTER AND MS-DOS COMPATIBLES.

STANDARD FEATURES INCLUDE:

- 79 STANDARD
- DIRECT I/O ACCESS
- FULL ACCESS TO MS-DOS FILES AND FUNCTIONS
- ENVIRONMENT SAVE & LOAD
- MULTI-SEGMENTED FOR LARGE APPLICATIONS
- EXTENDED ADDRESSING
- MEMORY ALLOCATION CONFIGURABLE ON-LINE
- AUTO LOAD SCREEN BOOT
- LINE & SCREEN EDITORS
- DECOMPILER AND DEBUGGING AIDS
- 8088 ASSEMBLER
- GRAPHICS & SOUND
- NGS ENHANCEMENTS
- DETAILED MANUAL
- INEXPENSIVE UPGRADES
- NGS USER NEWSLETTER

A COMPLETE FORTH DEVELOPMENT SYSTEM.

PRICES START AT $70

NEW◆HP-150 & HP-110 VERSIONS AVAILABLE

NEXT GENERATION SYSTEMS
P.O.BOX 2987
SANTA CLARA, CA. 95055
(408) 241-5909

PHASE ANGLE DIFFERENCE ANALYZER

C.H. TING - SAN MATEO, CALIFORNIA

The Phase Angle Difference Analyzer is an instrument which can determine accurately the time delay and phase difference between two analog signals sampled simultaneously by two sensors. The signals are digitized by two fast A/D converters and the results are analyzed by an NC4000 microprocessor. The frequency range of the input signals is from 20 Hz to 20 KHz, and the accuracy is about 0.05 degrees. It is especially useful for direction-finding based on acoustic waves.

Introduction

The Phase Angle Difference Analyzer (PANDA) measures the phase difference between two channels of analog inputs, assuming that the inputs are generated by the same source with different time delays between the two channels. This method is very similar to correlation analysis, but much simpler and faster.

Figure One shows a block diagram of the PANDA system. The signals received by two identical sensors are amplified and digitized by two analog-to-digital (A/D) converters. The digital outputs of the A/D converters are then fed into a microprocessor. The microprocessor analyzes the two channels of signal and determines the phase angle difference between the two channels.

A large number of samples are taken by the A/D converters and are stored in two arrays in the computer's memory. The time difference between the two input channels is computed by shifting and comparing values in the two arrays. The point of minimal difference is then interpolated and used to compute the phase difference. This analysis procedure is similar to a correlation analysis. However, the comparing step involves only subtraction, absolution, and addition. By avoiding multiplication, as required in conventional correlation analysis, the computation can be greatly accelerated to allow the PANDA system to perform phase measurements in real time.

The most crucial components in the PANDA system are the A/D converters, since their sampling rate determines the upper frequency of PANDA operation. It is assumed that the microprocessor can read the output from the A/D converters and store the sample data in memory. The upper frequency limit of PANDA was 20 KHz. To analyze an input signal of 20 KHz, PANDA must sample at a rate of 400 KHz so that it can determine the phase difference to the order of 0.1 degree.

PANDA must sample at a rate of 400 KHz.

Most commercial microprocessors, including rather sophisticated 16-bit machines like the Intel 8086 and Motorola 68000, cannot read data at this required sampling rate. Most data acquisition systems built for commercial microcomputers have an upper sampling rate of about 20 KHz and are not suitable for PANDA. A special CMOS 16-bit microprocessor, the NC4000[1] invented by Mr. Charles H. Moore, was thus chosen as the CPU for this PANDA. An NC4000 running at a clock of 4 MHz can perform 16-bit input/output at 4 MHz. To store data obtained from the I/O port takes two clock cycles. It is thus potentially capable of acquiring data at a speed exceeding 1 MHz. Two high-speed flash A/D converters, National ADC 0820, are used to digitize simultaneously the two input channels. A maximum conversion rate of an ADC 0820 is 1 MHz, which matches rather well with the NC4000.

The required sampling rate depends upon the frequency of the input signal. Generally, it is necessary to sample at least 20 times within one wave period. The total number of samples collected into memory is limited to 120 pairs due to memory and real-time processing requirements and limitations. Too few samples within one period would cause aliasing, and thus limit the range of angles in which the correct phase difference can be computed. Too many samples within one period would increase the error of measurement because the sampled array might not cover enough periods to compensate for the truncation of waves at the end of the sampling array. About ten periods are needed to give good accuracy.

Phase Difference Analysis

As mentioned above, the method adopted in PANDA to analyze the phase difference between the two input waves is closely related to the conventional correlation analysis. Figure Two shows schematically how the PANDA analysis is carried out. Two channels of input waves are sampled by the A/D converters and the sampled data are stored in two memory arrays, A and B. The data are represented as:

$A_0\ A_1\ A_3 \ldots A_{119}$
$B_0\ B_2\ B_3 \ldots B_{119}$

From these two sets of data, 20 phase-difference sums (bucket values) are computed. The bucket values are equivalent to the results of a correlation analysis:

$S_0\ S_1\ S_2 \ldots S_{19}$
$S_i = \mathrm{SUM}\ |A_j - B_{j+1-10}|$
$j=0 \text{ to } 109$

HARVARD SOFTWORKS

NUMBER ONE IN FORTH INNOVATION

(513) 748-0390 P.O. Box 69, Springboro, OH 45066

MEET THAT DEADLINE ! ! !

- Use subroutine libraries written for other languages! More efficiently!
- Combine raw power of extensible languages with convenience of carefully implemented functions!
- Yes, it is faster than optimized C!
- Compile 40,000 lines per minute!
- Stay totally interactive, even while compiling!
- Program at any level of abstraction from machine code thru application specific language with equal ease and efficiency!
- Alter routines without recompiling!
- Use source code for 2500 functions!
- Use data structures, control structures, and interface protocols from any other language!
- Implement borrowed feature, often more efficiently than in the source!
- Use an architecture that supports small programs or full megabyte ones with a single version!
- Forget chaotic syntax requirements!
- Outperform good programmers stuck using conventional languages! (But only until they also switch.)

HS/FORTH with FOOPS - The only flexible full multiple inheritance object oriented language under MSDOS!

Seeing is believing, OOL's really are incredible at simplifying important parts of any significant program. So naturally the theoreticians drive the idea into the ground trying to bend all tasks to their noble mold. Add on OOL's provide a better solution, but only Forth allows the add on to blend in as an integral part of the language and only HS/FORTH provides true multiple inheritance & membership.

Lets define classes BODY, ARM, and ROBOT, with methods MOVE and RAISE. The ROBOT class inherits:

```
  INHERIT> BODY
  HAS> ARM RightArm
  HAS> ARM LeftArm
```

If Simon, Alvin, and Theodore are robots we could control them with:

```
Alvin 's RightArm RAISE          or:
+5 -10 Simon MOVE                or:
+5 +20 FOR-ALL ROBOT MOVE
```

Now that is a null learning curve!

WAKE UP ! ! !

Forth is no longer a language that tempts programmers with "great expectations", then frustrates them with the need to reinvent simple tools expected in any commercial language.

HS/FORTH Meets Your Needs!

Don't judge Forth by public domain products or ones from vendors primarily interested in consulting - they profit from not providing needed tools! Public domain versions are cheap - if your time is worthless. Useful in learning Forth's basics, they fail to show its true potential. Not to mention being s-l-o-w.

We don't shortchange you with promises. We provide implemented functions to help you complete your application quickly. And we ask you not to shortchange us by trying to save a few bucks using inadequate public domain or pirate versions. We worked hard coming up with the ideas that you now see sprouting up in other Forths. We won't throw in the towel, but the drain on resources delays the introduction of even better tools. Don't kid yourself, you are not just another drop in the bucket, your personal decision really does matter. In return, we'll provide you with the best tools money can buy.

The only limit with Forth is your own imagination!

You can't add extensibility to fossilized compilers. You are at the mercy of that language's vendor. You can easily add features from other languages to **HS/FORTH**. And using our automatic optimizer or learning a very little bit of assembly language makes your addition zip along as well as in the parent language.

Speaking of assembly language, learning it in a supportive Forth environment turns the learning curve into a light speed escalator. People who failed previous attempts to use assembly language, conquer it in a few hours or days using HS/FORTH.

HS/FORTH runs under MSDOS or PCDOS, or from ROM. Each level includes **all** features of lower ones. Level upgrades: $25. plus price difference between levels. Sources code is in ordinary ASCII text files.

All HS/FORTH systems support full megabyte or larger programs & data, and run faster than any 64k limited ones even without automatic optimization -- which accepts almost anything and accelerates to near assembly language speed. Optimizer, assembler, and tools can load transiently. Resize segments, redefine words, eliminate headers without recompiling. Compile 79 and 83 Standard plus F83 programs.

STUDENT LEVEL **$145.**
text & scaled/clipped graphics in bit blit windows,mono,cga,ega,vga, fast ellipses, splines, bezier curves, arcs, fills, turtles; powerful parsing, formatting, file and device I/O; shells; interrupt handlers; call high level Forth from interrupts; single step trace, decompiler; music; compile 40,000 lines per minute, stacks; file search paths; formats into strings.

PERSONAL LEVEL **$245.**
software floating point, trig, transcendental, 18 digit integer & scaled integer math; vars: A B * IS C compiles to 4 words, 1..4 dimension var arrays; automatic optimizer-machine code speed.

PROFESSIONAL LEVEL **$395.**
hardware floating point - data structures for all data types from simple thru complex 4D var arrays - operations complete thru complex hyperbolics; turnkey, seal; interactive dynamic linker for foreign subroutine libraries; round robin & interrupt driven multitaskers; dynamic string manager; file blocks, sector mapped blocks; x86&7 assemblers.

PRODUCTION LEVEL **$495.**
Metacompiler: DOS/ROM/direct/indirect; threaded systems start at 200 bytes, Forth cores at 2 kbytes; C data structures & struct+ compiler; TurboWindow-C MetaGraphics library, 200 graphic/window functions, PostScript style line attributes & fonts, viewports.

PROFESSIONAL and PRODUCTION LEVEL EXTENSIONS:

FOOPS+ with multiple inheritance **$ 75.**
286FORTH or **386FORTH** **$295.**
16 Megabyte physical address space or gigabyte virtual for programs and data; DOS & BIOS fully and freely available; 32 bit address/operand range with 386.
BTRIEVE for HS/FORTH (Novell) **$199.**
ROMULUS HS/FORTH from ROM **$ 95.**
FFORTRAN translator/mathpak **$ 75.**
Compile Fortran subroutines! Formulas, logic, do loops, arrays; matrix math, FFT, linear equations, random numbers.

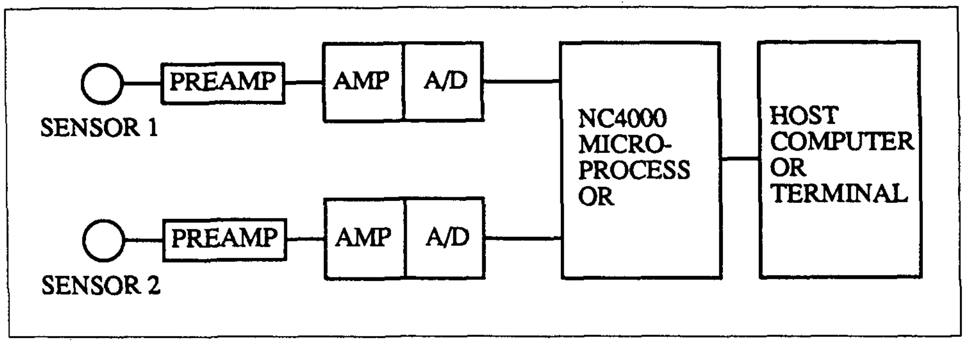


Figure One. Block diagram of PANDA system.

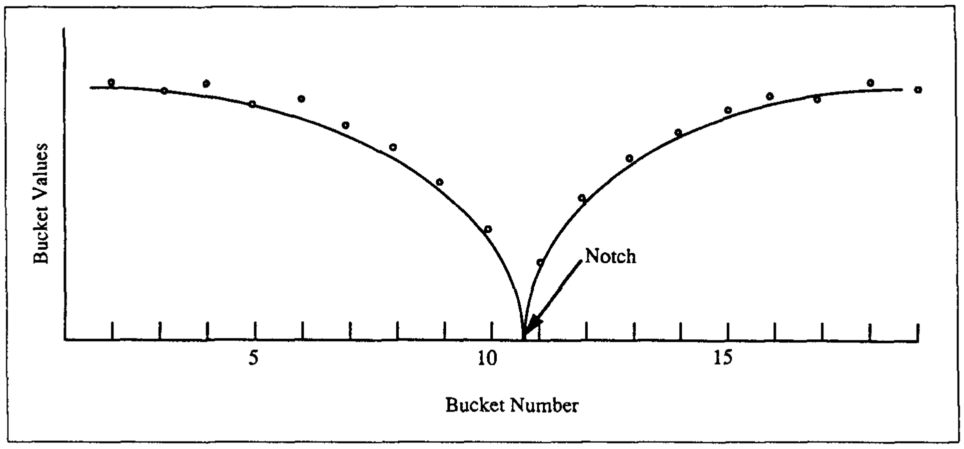


Figure Two. Analysis of phase difference.

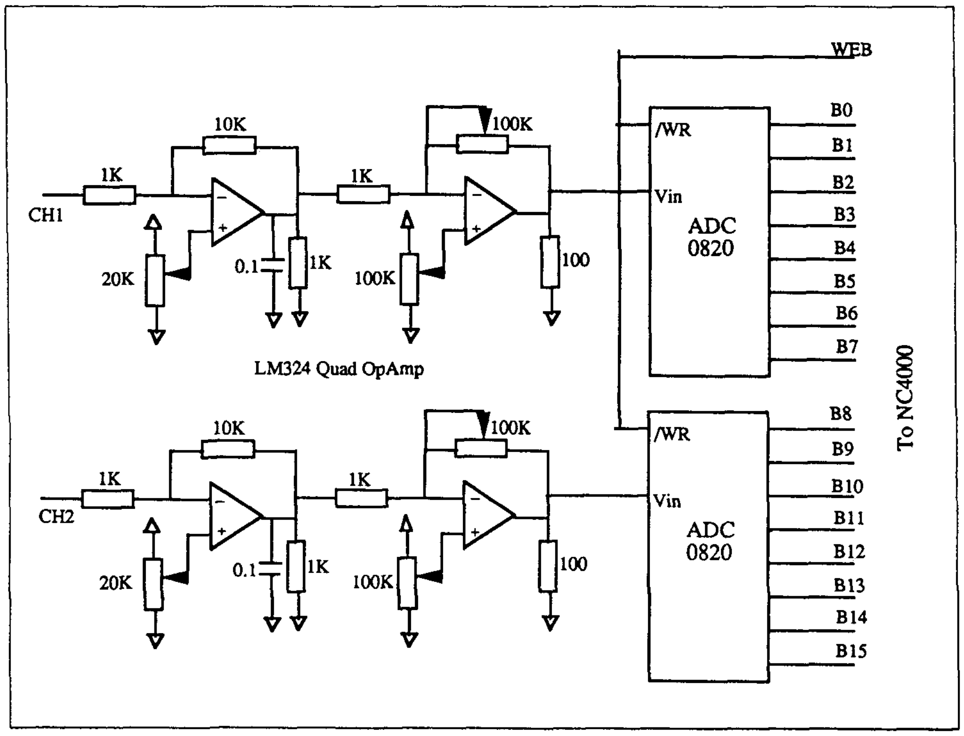


Figure Three. Analog and A/D circuitry in PANDA system.

If the two channels are identical, it is obvious that S_{10} should be zero and that it is the smallest of the 20 bucket values, because all the difference terms in this sum are zero. Values neighboring S_{10} should increase gradually, forming a notch at S_{10} when the bucket values are plotted against the bucket number. The phase difference between the two input channels can be determined accurately by how much the notch shifts away from the S_{10}, as shown in Figure Two.

A simple interpolation algorithm is applied to determine the true notch-bucket number, which is related to the time difference of signal arrival at the two sensors. If the *i*th bucket has the lowest bucket value S_i, the true notch is calculated from the following equation:

```
NOTCH = i + (S_{i-1} - S_{i+1})/
        2(S_{i-1} + S_{i+1} - 2S_i)
```

An equation for `BASELINE` is as follows:

```
BASELINE = D/(delta * V)
```

where D is the distance between the two sensors, delta is the sampling time, and V is the speed of signal. The angle-of-arrival (AOA) is then:

```
AOA = sin^-1 [(NOTCH - 10)/
      BASELINE]
```

In the present PANDA system, the sampling time of the A/D converter is programmable through the system variable `DELAY`. The relationship between `DELAY` and delta is:

```
delta = 0.25 (DELAY+11) µsec.
```

The NC4000 processor cycle is 0.25 microseconds using a 4 MHz clock. It takes 11 cycles to start the A/D converters, wait until the data is digitized, input the data, and store the data to memory.

Hardware

The PANDA main unit consisted of two circuit boards, one analog and one digital. The analog processing board contained a pair of OpAmp circuits to amplify and condition the input signals, and a pair of fast A/D converters to digitize the sensor input signal. The digital processing board contained the main CPU chip (NC4000), EPROM, and SRAM memory chips, and digital I/O circuitry. The CPU analyzes the input signals and sends the results to a terminal (Qume QVT-102) or to a host computer for archiving and displaying tasks.

Figure Three shows the schematics of the analog board. The amplifier/signal-conditioners are constructed with a single quad OpAmp IC (National LM324). Each single channel uses two OpAmps, one for input conditioning with a gain of ten, and another for amplification and offset adjustment to present the signals optimized for the A/D converter. The amplifier stage has an adjustable gain of 0.1 to 100, and an offset range of 0–5 volts.

Two A/D converters are used in parallel to convert simultaneously the two channels of sensor signals in order to increase the conversion throughput and to avoid skewing in the sampling process.[2] The A/D converters are eight-bit half-flash converters. The start-conversion clock is provided by an I/O write-enable (WEB) signal from the MC4000, and the output data are read by the NC4000 through the B port. Each A/D provides eight bits of data to the 16-bit B port. Although the A/D converters and the SPU are capable of running up to a 1 MHz sampling rate, due to the start-conversion and memory-storing overhead in the data acquisition process, the maximum practical sampling rate is 400 KHz on both channels.

The digital processor board is a CMOS single-board microprocessor (Silicon Composers SC1000-CPU). It contains a very fast CMOS 16-bit microprocessor (Novix NC4000). This CPU executes one CPU instruction in every clock cycle derived from a 4 MHz single-phase clock. All branch and subroutine-call instructions are also completed in one clock cycle. Memory access requires two cycles. This speed allowed the PANDA to obtain and analyze data from the A/D converters at the rate required by the PANDA experiments.

Surrounding the NC4000 chip are two eight Kbyte EPROM memory chips hosting the PANDA software, two 32 Kbyte RAM chips for data storage, and four eight Kbyte RAM chips serving as one data stack and one return stack. A few MSI glue chips complete the processor board: a 74HC138 for memory decoding, a CD4050 for reset and serial I/O buffering, and a 4 MHz CMOS clock.

Two bits in the X I/O port on the NC4000 are used to emulate an RS-232 serial interface port, which allows the NC4000 to communicate with a terminal or host computer. Only the transmit and receive lines are used in this RS-232 interface. The data format is 9600 baud, eight data bits, and one stop bit. Parity is disabled. The transmit and receive lines are buffered through the CD4050.

The digital and analog boards are connected through a bussed backplane. Although all the memory and I/O signals are brought out to the backplane, the analog processor uses only the B port I/O signals.

Software

The software installed in the PANDA system controls the operations of the system and its user-host interface. It allows the user to specify the conditions under which the PANDA system is to be operated, such as the sampling rate, the time delay between data reports, and the format of the data reports. It also allows a host computer to receive the results and to process the raw data with more sophisticated data analysis

Total control with *LMI FORTH*™

For Programming Professionals: an expanding family of compatible, high-performance, compilers for microcomputers

For Development:
Interactive Forth-83 Interpreter/Compilers for MS-DOS, OS/2, and the 80386
- 16-bit and 32-bit implementations
- Full screen editor and assembler
- Uses standard operating system files
- 500 page manual written in plain English
- Support for graphics,floating point, native code generation

For Applications: Forth-83 Metacompiler
- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 8088, 68000, 6502, 8051, 8096, 1802, 6303, 6809, 68HC11,34010, V25, RTX-2000
- No license fee or royalty for compiled applications

Laboratory Microsystems Incorporated
Post Office Box 10430, Marina del Rey, CA 90295
Phone Credit Card Orders to: (213) 306-7412
FAX: (213) 301-0761

Thinking of using the Zilog Forth Super8 Chip?

Inner Access has FORTH Super8 development tools for you!

- **Development Lab**
- **Metacompiler**
- **Development ROMS**

with F83 based terminal emulation and file server software for the IBM

Inner Access Corporation
Box 888 Belmont CA 94002
(415) 591-8295

```
( PANDA SETUP, 10AUG86CHT )

HEX
3C0 CONSTANT RESULTS            ( Bucket values )
400 CONSTANT 1TEST              ( Primary data array )
A00 CONSTANT 2TEST              ( Second data array )
  0 CONSTANT COUNTER            ( Report counter )
  1 CONSTANT DELAY              ( Delay between samples )
  2 CONSTANT WAITING            ( Delay between reports )
  3 CONSTANT RADIUS             ( Baseline between sensors )
  4 CONSTANT SAMPLES            ( Number of A/D samples )
  5 CONSTANT FREQUENCY          ( Frequencey multiplier )
  6 CONSTANT PHASE              ( Phase offset )
  7 CONSTANT TICKS
DECIMAL


: ONE-PASS ( PHASE -- SUM )
     0                          ( Initial sum )
     SAMPLES @ 20 -             ( Analyze only 100 samples )
     FOR  OVER 1TEST + I + @    ( Channel 1 data with offset )
          2TEST I + 10 + @      ( Channel 2 data )
          - ABS +               ( Subtract, absolute, accumulate )
     NEXT
     SWAP DROP   ;



: 20-PASSES ( -- )
     1 COUNTER +!               ( Increment report count )
     RESULTS 20 ERASE           ( Clear sum array )
     19 FOR
          I ONE-PASS            ( Do analysis )
          I RESULTS + !         ( Store away resulting sums )
     NEXT ;


: 3.R ( N -- )                  ( For Radix 64 output )
     <# # # # #> ;


: SHOW-RESULTS ( -- )
     COUNTER @ 3.R              ( Count field )
     19 FOR
          RESULTS I + @ 3.R     ( 20 sums )
     NEXT
     CR 10 EMIT ;               ( CR and two LF's )

( PANDA ANALYSIS, 07MAY86CHT )


: RATIO ( N1 N2 N3 --- RATIO, N1>N2>N3 )
     SWAP OVER - >R             ( N2-N3 )
     - R> 500                   ( N1-N3 )
     ROT */                     ( [N2-N3]*500/[N1-N3] )


     500 SWAP - ;               ( [N2-N3+N1-N3]*500/[N1-N3] )
```

software.

Because NC4000 is a microprocessor supporting the high-level language Forth, it is natural that PANDA is programmed in Forth. The SC1000-CPU board developed by Silicon Composers includes the operating system cmForth[3] written by Charles H. Moore, the creator of Forth and also the chief designer of the NC4000. The PANDA program is generated by the target compiler in cmForth and installed in the PANDA system via a pair of eight Kbyte EPROMs.

The complete source code of PANDA is shown in Listing One. Very extensive comments are included and most of the code is self-explanatory. Only a few words are highlighted here to illustrate how the PANDA system is used to collect data and compute the phase difference between signals received from the input channels.

A set of user-changeable parameters is defined as user variables so that the PANDA system can be adapted to analyze input signals within a very wide frequency range:

`SAMPLES`	Number of samples colected in one acquisition process. (120)
`DELAY`	Number of empty loops inserted between samplings. (4)
`WAITING`	Number of 0.3-second delays between data reports. (10)
`RADIUS`	Bucket-number*1000, corresponding to a half period. (8000)
`FREQUENCY`	Number of periods in synthetic test input signals. (20)
`PHASE`	Phase delay between two channels in the `TEST` routine. (Variable)
`COUNTER`	A running counter for data reports. (Variable)

The numbers in parentheses are default values to analyze signals in a 20 KHz range.

`1TEST` is the primary data array to receive raw data from the A/D converters. `A/D` clocks the A/D converters and stores data into `1TEST`. Data from channel one is stored in the lower eight bits and data from channel two is stored in the upper eight bits of a 16-bit word in `1TEST`. `DIGEST` extracts channel two data and stores it in the

2TEST array while it clears the upper bytes in 1TEST.

ONE-PASS takes a sample from 1TEST, subtracts it from a corresponding sample in 2TEST, and then accumulates the absolute difference into one term in the RESULTS array. The sample in 2TEST is offset from the sample in 1TEST by a phase factor taken from the variable PHASE. When the phase factor is ten, the two arrays are exactly aligned. There are 20 sums in RESULTS, the phase difference between the two input signals is between -10 and +9 sampling intervals.

ONE-PASS is executed 20 times in 20-PASSES, with the phase factor varied from zero to 19. The RESULTS array then contains the sums of the phase difference analysis. These sums are referred to as buckets and bucket values. Plotting these bucket values against the phase factors, a curve with a notch near the center can be obtained (Figure Three). The minimum of the notch is the phase difference between the two input signals, relative to the center bucket with a phase factor of ten. The time delay between the two input signals is then the bucket value of the notch subtracted from ten and multiplied by the time interval between two consecutive samples.

MINIMUM scans through the RESULTS array and returns the bucket number of the bucket with the lowest bucket value. RATIO computes the ratio (N3-N2)/(N3-N1+N2-N1), where N3>N2>N1. This is the computation needed to interpolate among the three lowest buckets to determine the true position of the notch among the 20 buckets. NOTCH calls MINIMUM and RATIO to determine the notch position, which is represented by an integer bucket number and a fraction of the bucket number multiplied by 1000.

If the two sensors are measuring the signals from a single source, the angle-of-arrival (AOA) of the source relative to the sensors can be calculated with the knowledge of the phase difference of the signals arriving at the sensors, the distance between the sensors, and the velocity of the travelling signal. ARCSIN converts the phase difference to the angle-of-arrival by interpolation with the help of the arc-sine table in (ARCSIN). The value in RADIUS is used as the baseline for arc-sine calculation.

There are several data-reporting routines to sample the signals and display the

```
: MINIMUM ( -- N )
    19 RESULTS 19 + @          ( Bubble sort )
    18 FOR
         I RESULTS + @
         2DUP >
         IF   SWAP DROP
              SWAP DROP
              I SWAP
         ELSE DROP
         THEN
    NEXT
    DROP ;

: NOTCH ( -- REMAINDER BUCKET )
     MINIMUM DUP
     RESULTS + >R              ( Smallest bucket )
     I 1 + @                   ( Two neighboring buckets )
     I 1 - @
     2DUP >                    ( Reorder bucket values )
     IF   R> @ RATIO           ( and intrapolate between )
          1000 SWAP -          ( buckets )
          SWAP 1 -
     ELSE SWAP R> @ RATIO
          SWAP
     THEN   ;


( DATA ACQUISITION, 10AUG86CHT )

HEX

: A/D ( GET SAMPLES INTO 1TEST ARRAY )
     0 1TEST 1 - ( DATA ADDR )
     SAMPLES @
     FOR  0 8 I!               ( Start A/D conversion )
          1 ( DELAY )
          @ FOR NEXT           ( Wait till data ready )
          1 !+                 ( Store away previous data )
          8 I@ SWAP            ( Read both channels )
     NEXT
     2DROP ;

: DIGEST                       ( Separate CH2 data )
     SAMPLES @ 1 -
     FOR
          1TEST I + DUP @      ( Get stored data )
          DUP >R
          FF AND SWAP !        ( Low byte to 1TEST )
          R> 6 TIMES           ( Shift high byte )
          [ 8001 , ( 2/ ) ]
          FF AND 2TEST I + !   ( Store CH2 data to 2TEST )
     NEXT ;


 DECIMAL

 : ACQUISITION ( -- )
      A/D DIGEST 20-PASSES ;
```

```
( SIMULATED DATA, 18AUG86CHT )

7168 CONSTANT SINE              ( A table of sine function )

: PATTERN ( FREQUENCY PHASE -- )
    SAMPLES @
    FOR  I FREQUENCY @          ( Get frequency multiplier )
         * DUP 1023 AND         ( Offset into sine table )
         SINE + @               ( Get data from table )
         1TEST I + !            ( Put into 1TEST array )
         PHASE @ + 1023 AND     ( Add phase offset )
         SINE + @               ( Offset data )
         2TEST I + !            ( Put to 2TEST array )
    NEXT ;

: ?KEY ( -- F )                 ( Test RS232 input line )
    0   WAITING @
    FOR
         20000 FOR
              RX 16 XOR OR
         NEXT
    NEXT ;

: TEST ( -- )                   ( Test PANDA with sine waves )
    64 BASE !
    0 COUNTER !
    PHASE @ DUP >R
    0 PHASE !
    BEGIN
         PATTERN                ( Synthesize data )
         20-PASSES              ( Analyze )
         SHOW-RESULTS           ( Report results )
         DUP PHASE +!
    ?KEY UNTIL
    DROP R> PHASE !
    ;

( intrapolation  16aug86cht )

CREATE (ARCSIN)  ( a table of function values )
    0 , 500 , 1002 , 1506 , 2014 ,
    2526 , 3046 , 3576 , 4116 ,
    4668 , 5240 , 5824 , 6434 ,
    7076 , 7754 , 8480 , 9272 ,
    10160 , 11198 , 12532 , 15708 ,

: ARCSIN ( 10000*SIN -- 100*ARCSIN )
    DUP >R   ABS

    10000 MIN   500 /MOD        ( 2* )
    (ARCSIN) + 2@               ( Intrapolate )
    DUP >R   -
    500 */   R> +
    9000 15708 */               ( Scale to 90 degrees )
    R>   0<
    IF NEGATE THEN   ;

: ANGLE ( FRACTION BUCKET -- ANGLE*100 )
    10 - 1000 * +
    10000 RADIUS @ */           ( Scale to the baseline )
    ARCSIN   ;
```

results. DIRECTION sends a continuous stream of AOA values to be displayed on a terminal. The AOA values have a range from -9000 to 9000, corresponding to -90 and 90 degrees. It can be interrupted by pressing a key on the terminal. METER is a visual display routine, showing a vertical bar among 80 columns on a CRT terminal. 80 columns allow the display of AOA results with a resolution of about 2.5 degrees in a -90 to 90 degree field.

RUN is used to send the raw bucket values to a host computer. To minimize the transmission time, bucket values are encoded in Radix 64. Each record contains 66 characters. The first three characters encode a record number, the next 60 characters encode 20 bucket values, and the last three characters CR-LF-LF terminate a record. The Radix 64 scheme allows a 16-bit value to be represented by three ASCII characters without any ambiguity.

When the PANDA system is turned on, it enters the RUN procedure immediately and sends the Radix 64 reports continuously. Any keystroke will terminate RUN, and the user can operate it interactively. The RESET procedure boots PANDA from EPROMs.

Results and Discussion

The PANDA system works very accurately and the response is very fast. Extensive experiments in calibration and also in analyzing real signals have shown that it performs well between 20 Hz and 20 KHz. The accuracy and reproducibility most often depend upon the noise contents in the input signals. When the input signals are strong and well balanced, the phase angle difference can be determined down to 0.05 degrees. PANDA was used to analyze audio and underwater acoustic signals in direction-finding applications.

A very interesting property of PANDA is that its performance does not depend on the wave shape on the input signals. Sine waves, square waves, and randomly shaped waves can be analyzed with the same degree of ease and accuracy. The performance of PANDA degrades gracefully with increased noise in the input signals and imbalance between the two channels. In very noisy environments, the accuracy of the phase difference measurements can be increased by temporal integration.

It is difficult to theorize the PANDA methodology, because of the difficulty in modelling the absolution of difference between two signal channels. Correlation analysis is related to the power spectrum of the signals, while the PANDA analysis is more closely related to the amplitude spectrum. The PANDA analysis is likely to yield more accurate results in phase difference, in which all the signal points contribute equally. On the other hand, the results in correlation analysis are weighed more heavily towards signal points of higher amplitude.

Since the PANDA method uses only addition, subtraction, and absolution, the computation load to the controlling microprocessor is much less than that of correlation analysis. The required dynamic range of the sums is also much smaller because it eliminates the multiplication operations. Consequently, a 16-bit microprocessor works comfortably. Even double integers are not necessary in the computation.

The maximum frequency range of PANDA can be pushed to about 500 KHz using a 10 MHz RTX 2000 and a 10 MHz A/D converter. Both are readily available now. Most of the recent crop of fast-flash A/D converters can be driven directly by reading the strobes from the microprocessor. Most of the instructions in the `A/D` procedure to strobe the converter can then be eliminated and the analysis can run much faster.

References

1. NC4000 is a microprocessor manufactured by Novix, Inc. in San Jose, California. However, Novix, Inc. is no longer actively supporting this product. A limited quantity of this chip is still available through Silicon Composers in Palo Alto, California.

2. C.H. Ting, "A/D Converters with the NC4000," *More on NC4000*, Vol. 3, p. 83, 1987.

3. C.H. Ting, *Footsteps in an Empty Valley*, 3rd ed., pp. 83–147. Offete Enterprises, Inc., 1988. This book contains the most detailed information about the NC4000 itself and its operating system, cmForth.

```
( CONTROL LOOPS, 31DEC86CHT )

: RUN ( -- )                    ( Default output )
    64 BASE !
    BEGIN
         ACQUISITION
         SHOW-RESULTS
    ?KEY UNTIL
    ;

: DIRECTION ( -- )
    DECIMAL
    BEGIN
         ACQUISITION
         NOTCH ANGLE            ( Show AOA )   .
    ?KEY UNTIL ;

: SCALE ( -- )
    CR 18 FOR
         I 9 - ABS
         48 + EMIT              ( Show bar graph scale )
         2 FOR 46 EMIT NEXT
    NEXT CR   ;


: METER ( -- )
    BEGIN
         ACQUISITION
         NOTCH ANGLE            ( Computer angle )
         COUNTER @ 15 AND       ( Show scale occasionally )
         IF CR ELSE SCALE THEN
         9000 + 250 /           ( Scaling to 80 columns )
         71 FOR
              DUP I -
              IF 32 ELSE 124 THEN
              EMIT              ( Display needle point )
         NEXT
         DROP
    ?KEY UNTIL ;

: RESET ( INITIALIZE FOR 10 KHZ OPERATION)

  BOOT                          ( Initialize NC4000 )
  0 COUNTER !                   ( Initialize PANDA variables )
  9 DELAY !                     ( for 20 KHz measurements )
  10 WAITING !
  8000 RADIUS !
  20 FREQUENCY !
  120 SAMPLES !
  20 TICKS !
  0 9 I! 0 10 I! 0 11 I!        ( Initialize B port )
  RUN QUIT ;                    ( Run host reporting routine)
```

ANS FORTH
HARDWARE INDEPENDENCE

JOHN R. HAYES - LAUREL, MARYLAND

Forth has always worked closely with the underlying hardware. The most popular architectures used to implement Forth have had byte-addressed memory, 16-bit operations, and two's-complement number representation. The Forth-83 Standard dictates that these particular features must be present in a Forth-83 Standard system and that Forth-83 programs may exploit these features freely. However, there are many beasts in the architectural jungle that are bit addressed or cell addressed, or prefer 32-bit operations, or represent numbers in one's complement or BCD. Since one of Forth's strengths is its usefulness in "strange" environments on "unusual" hardware with "peculiar" features, it is important that a standard Forth run on these machines too.

A primary goal of the ANS Forth standard is to increase the types of machines that can support a standard Forth. This is accomplished by allowing some key Forth terms to be implementation defined (i.e., how big is a cell?) and by providing Forth operators (words) that conceal the implementation. This frees the implementor to produce the Forth system that most effectively utilizes the native hardware. The machine-independent operators, together with some programmer discipline, enable a programmer to write Forth programs that work on a wide variety of machines.

The ANS Forth standard cannot and should not force anyone to write a portable program. In situations where performance is paramount, the programmer is encouraged to use every trick in the book. Writing a portable program is an opportunity. If a Forth programmer invents a new programming technique, then implements it so that it relies on every quirk of his Forth system, that program is only useful to people with an identical system. A portable program benefits a greater number of people and is consequently more valuable. When programming for profit, a portable program automatically has a larger potential market than a non-portable program.

The computers that can host ANS Forth form a superset of the machines that run Forth-83. Forth-83 programs will work, with very little modification, on ANS Forth systems that use 16-bit cells and address memory as eight-bit bytes. However, Forth-83 programs will probably need substantial modification to run on other ANS Forth systems (e.g., systems with 32-bit cells). In other words, non-portable programs remain non-portable. Increasing the range of machines that can support ANS Forth does not diminish the range of machines that run Forth-83 programs. This is important to remember while studying the ANS Forth definitions of such "familiar" concepts as byte, cell, and memory addressing. The definitions were carefully chosen to be a generalization of Forth-83's definitions. Forth-83 programs (in ANS Forth jargon) have an environmental dependency that cells be 16-bits wide and that memory is addressed as eight-bit bytes.

Systems with different cell sizes will be encountered...

The rest of this article describes some ANS Forth features for making a program independent of hardware peculiarities. It is difficult for someone familiar with only one machine architecture to imagine the problems caused by transporting programs between dissimilar machines. Consequently, examples of specific architectures with their respective problems are given.

Hardware Independence

Data and memory are the stones and mortar of program construction. Unfortunately, each computer treats data and memory differently. The ANS Forth Programming Systems standard gives definitions of data and memory that apply to a wide variety of computers. These definitions give us a way to talk about the common elements of data and memory while ignoring the details of specific hardware. Similarly, ANS Forth programs that use data and memory in ways that conform to these definitions can also ignore hardware details. The following sections discuss the definitions and describe how to write programs that are independent of the data and memory peculiarities of different computers.

Definitions

Three terms defined by ANS Forth are address unit, cell, and byte. The address space of an ANS Forth system is divided into an array of address units; an address unit is the smallest collection of bits that can be addressed. In other words, an address unit is the number of bits spanned by the addresses *addr* and *addr+1*. The most prevalent machines use eight-bit address units. Such "byte-addressed" machines include the Intel 8086 and Motorola 68000 families. However, other address unit sizes exist. There are machines that are bit addressed and machines that are 4-bit-nibble addressed. There are also machines with address units larger than eight bits. For example, several Forth-in-hardware computers are cell addressed (Novix NC4016 with 16-bit address units and the Silicon Composers' SC32 with 32-bit address units).

The cell is the fundamental data type of a Forth system. A cell can be a single-precision integer or a memory address. Forth's parameter and return stacks are stacks of cells. Forth-83 specifies that a cell is 16 bits; in ANS Forth the size of a cell is an implementation-defined number of address units. Thus, an ANS Forth implemented on a 16-bit microprocessor could use a 16-bit cell, and an implementation on a 32-bit machine could use a 32-bit cell. 18-bit machines (PDP-15), 36-bit machines (PDP-10), etc. could also support ANS Forth systems with 18- or 36-bit cells, respectively. In all of these systems, `DUP` does the same thing: it duplicates the top of the parameter stack. `!` (store) behaves consistently too: given two cells on the parameter stack, it stores the second cell in the memory location designated by the top cell.

Historically, the definition of a byte has been the most convenient amount of storage that could hold a character. The majority of machines built in recent years use eight-bit address units and store one character per address unit. This has resulted in the widespread assumption that a byte is always eight bits. ANS Forth uses the more general definition: a byte is an implementation-defined number of address units (but at least eight bits) used to hold a character. This removes the need for a Forth implementor to provide eight-bit bytes on processors where it is inappropriate. For example, on an 18-bit machine with a nine-bit address unit, a nine-bit byte would be most convenient. Since, by definition, you can't address anything smaller than an address unit, a byte must be at least as big as an address unit. This will result in big bytes on machines with large address units. An example is a 16-bit-cell addressed machine, where a 16-bit byte makes the most sense.

Addressing Memory

ANS Forth eliminates many portability problems by using the above definitions. One of the most common portability problems is addressing successive cells in memory. Given the memory address of a cell, how do you find the address of the next cell? In Forth-83 this is easy: `2 +`. This code assumes that memory is addressed in eight-bit units (octets) and that a cell is 16-bits wide. On an octet-addressed machine with 32-bit cells, the code to find the next cell would be `4 +`. The code would be `1+` on a cell-addressed processor and `16 +` on a bit-addressed processor with 16-bit cells. ANS Forth provides a next-cell operator named `CELL+` that can be used in all of these cases. Given an address, `CELL+` adjusts the address by the size of a cell (measured in address units).

A related problem is that of addressing an array of cells in an arbitrary order. A defining word to create an array of cells using Forth-83 would be:

```
: ARRAY CREATE
  2* ALLOT DOES>
  SWAP 2* + ;
```

Use of `2*` to scale the array index assumes octet addressing and 16-bit cells again. As in the example above, different versions of the code would be needed for different machines. ANS Forth provides a portable scaling operator named `CELLS` . Given a number *n*, `CELLS` returns the number of address units needed to hold *n* cells. A portable definition of `ARRAY` is:

```
: ARRAY CREATE
  CELLS ALLOT DOES>
  SWAP  CELLS + ;
```

There are also portability problems with addressing arrays of bytes. In Forth-83 (and in the most common ANS Forth implementations), the size of a byte will equal the size of an address unit. Consequently, addresses of successive bytes in memory can be found using `1+` and scaling indices into a byte array is a no-op (i.e., `1 *`). However, there are cases where a byte is larger than an address unit. Examples include systems with small address units (e.g., bit- and nibble-addressed systems) and systems with large character sets (e.g., 16-bit characters on an octet-addressed machine). `BYTE+` and `BYTES` operators, analogous to `CELL+` and `CELLS`, are available to allow maximum portability.

ANS Forth generalizes the definitions of some Forth words that operate on chunks of memory to use address units. One example is `ALLOT` . By prefixing `ALLOT` with the appropriate scaling operator (`CELLS` , `BYTES`, etc.), space for any desired data structure can be allocated (see definition of `ARRAY` above). For example:

```
CREATE ABUFFER
5 BYTES ALLOT
```
(Allots a 5-byte buffer.)

The memory-block move word also uses address units:

```
source dest 8 CELLS MOVE
```
(Moves eight cells.)

MAKE YOUR SMALL COMPUTER
THINK BIG
(We've been doing it since 1977 for IBM PC, XT, AT, PS2, and TRS-80 models 1, 3, 4 & 4P.)

FOR THE OFFICE — Simplify and speed your work with our outstanding word processing, database handlers, and general ledger software. They are easy to use, powerful, with executive-look print-outs, reasonable site license costs and comfortable, reliable support. Ralph K. Andrist, author/historian, says: "FORTHWRITE lets me concentrate on my manuscript, not the computer." Stewart Johnson, Boston Mailing Co., says: "We use DATAHANDLER-PLUS because it's the best we've seen."

MMSFORTH System Disk	from $179.95

Modular pricing — Integrate with System Disk only what you need:

FORTHWRITE - Wordprocessor	$99.95
DATAHANDLER - Database	$59.95
DATAHANDLER-PLUS - Database	$99.95
FORTHCOM - for Communications	$49.95
GENERAL LEDGER - Accounting System	$250.00

FOR PROGRAMMERS — Build programs FASTER and SMALLER with our "Intelligent" MMSFORTH System and applications modules, plus the famous MMSFORTH continuing support. Most modules include source code. Ferren MacIntyre, oceanographer, says: "Forth is the language that microcomputers were invented to run."

SOFTWARE MANUFACTURERS — Efficient software tools save time and money. MMSFORTH's flexibility, compactness and speed have resulted in better products in less time for a wide range of software developers including Ashton-Tate, Excalibur Technologies, Lindbergh Systems, Lockheed Missile and Space Division, and NASA-Goddard.

MMSFORTH V2.4 System Disk	from $179.95

Needs only 24K RAM compared to 100K for BASIC, C, Pascal and others. Convert your computer into a Forth virtual machine with sophisticated Forth editor and related tools. This can result in 4 to 10 times greater productivity.

Modular pricing — Integrate with System Disk only what you need:

EXPERT-2 - Expert System Development	$69.95
FORTHCOM - Flexible data transfer	$49.95
UTILITIES - Graphics, 8087 support and other facilities.	

and a little more!

THIRTY-DAY FREE OFFER — Free MMSFORTH GAMES DISK worth $39.95, with purchase of MMSFORTH System. CRYPTOQUOTE HELPER, OTHELLO, BREAKFORTH and others.

MMSFORTH

MILLER MICROCOMPUTER SERVICES
61 Lake Shore Road, Natick, MA 01760
(508/653-6136, 9 am - 9 pm)

Call for free brochure, technical info or pricing details.

Alignment Problems

Not all addresses are created equal. Many processors have restrictions on the addresses that can be used by memory access instructions. For example, on a Motorola 68000, 16-bit or 32-bit data can be accessed only at even addresses. Another example is Sun's SPARC architecture, where 16-bit data can be loaded or stored only at even addresses and 32-bit data only at addresses that are multiples of four.

An implementor of ANS Forth can handle these alignment restrictions in one of two ways. Forth's memory access words (`@`, `!`, `+!`, etc.) could be implemented in terms of smaller-width access instructions which have no alignment restrictions. For example, on a 68000 Forth with 16-bit cells, `@` could be implemented with two 68000 byte-fetch instructions and a reassembly of the bytes into a 16-bit cell. Although this conceals hardware ugliness from the programmer, it is inefficient. An alternate implementation of ANS Forth could define each memory access word using the native instructions that most closely match the word's function. On a 68000 Forth with 16-bit cells, `@` would use the 68000's 16-bit move instruction. In this case, responsibility for giving `@` a correctly aligned address devolves onto the programmer. A portable ANS Forth program must assume the worst and use the alignment operators described below.

One of the most common problems caused by alignment restrictions is in creating tables containing both bytes and cells. When initializing the table using `,` and `C,` data is stored at the end of the dictionary. Consequently, the dictionary pointer must be suitably aligned. For example, a non-portable table definition would be:

```
CREATE ATABLE
1 C, X , 2 C, Y ,
```

On the second 68000 Forth implementation described above, `CREATE` would leave the dictionary pointer at an even address, the `1 C,` would make the dictionary pointer odd, and `,` would crash the system by storing `X` at an odd address. A portable way to create the table is:

```
CREATE ATABLE
1 C, ALIGN X ,
2 C, ALIGN Y ,
```

(Continued on page 33.)

Listing One

```
\ Structure access words usage:
\   structure foo                      \ Declare a structure
\        3 bytes: .part1               \ consisting of a 3-byte part,
\           cell: .part2               \ a one-cell part,
\           byte: .part3               \ and a one-byte part.
\   endstructure
\
\   structure foobar                   \ Declare another structure
\        2 cells: .this                \ consisting of two cells,
\     foo struct: .that                \ and substructure
\   endstructure
\
\   create teststruct
\   foobar allot                       \ Allocate a structure instance
\     123 teststruct
\     .that .part2 !                   \ & store something in it.
```

Implementation notes:

1. Structure instances must be put at an aligned address (i.e., via CREATE).
2. ENDSTRUCTURE pads out the end of the structure—this is unnecessary.

```
: structure     ( - pfa template )
  \ Start structure declaration.
    create here 0 , 0
    does> @ ;   ( addr[size] - size )

: aus:          ( offset size - offset' )
  \ Structure member compiler.
    create over , +
    does> @ + ;  ( base addr[offset] - base' )
  \ Add member's offset to base.

: bytes:        ( template n - template' )
  \ Create n byte member.
    bytes aus: ;

: byte:         ( template - template' )
  \ Create 1 byte member.
    1 bytes: ;

: cells:        ( template n - template' )
  \ Create n cell member.
    cells >r realign r> aus: ;

: cell:         ( template - template' )
  \ Create 1 cell member.
    1 cells: ;

: struct:       ( template size - template' )
  \ Create member of given size.
    >r realign r> aus: ;

: endstructure  ( pfa template - )
  realign swap ! ;
```

SILICON COMPOSERS

Performance, Quality, Service

SC/FOX PCS32 Parallel Coprocessor System32

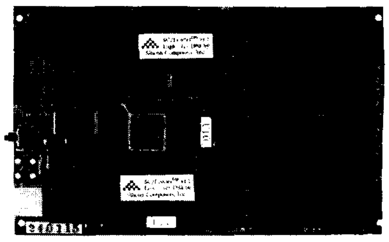
Uses the 32-bit SC32tm Forth CPU.
System speed options: 8 or 10 MHz.
Full-length 8- or 16-bit PC/XT/AT plug-in board.
64K to 1M byte, 0-wait-state static RAM.
Hardware expansion, two 50-pin strip headers.
Includes SC/Forth32, based on the Forth-83 Standard.

SC/FOX PCS Parallel Coprocessor System

Uses Harris RTX 2000tm real-time Forth CPU.
System speed options: 8, 10 or **12 MHz.**
Full-length 8- or 16-bit PC/XT/AT plug-in board.
32K to 1M bytes, 0-wait-state static RAM.
Hardware expansion, two 50-pin strip headers.
Includes FCompiler; SC/Forth optional.

SC/FOX SBC Single Board Computer

Uses RTX 2000 real-time Forth CPU.
System speed options: 8, 10, 12 or **14 MHz.**
32K to 512K bytes 0-wait-state static RAM.
RS232 56K-baud serial and printer ports.
Hardware expansion, two 50-pin strip headers.
64K bytes of shadow-EPROM space.
Eurocard size: 100mm by 160mm.
Includes FCompiler; optional SC/Forth EPROM.



SC/FOX SCSI I/O Daughter Board

Plug-on daughter board for SC/FOX PCS and SBC.
Source s/w drivers for FCompiler and SC/Forth.
SCSI adaptor with 5 Mbytes/sec synchronous or
3 Mbytes/sec asynchronous transfer rates.
Floppy disk adaptor; up to 4 drives, any type.
Full RS-232C Serial Port, 50 to 56K Baud.
16-bit bidirectional, latching parallel port.

SC/Forthtm Language

Based on the Forth-83 Standard.
15-priority time-sliced multitasking.
Supports user-defined PAUSE.
Automatic optimization and μcode support.
Turnkey application support.
Extended structures and case statement.
Double number extensions.
Infix equation notation option.
Block or text file interpretation.
Optional source-code developer system.
Supports program spawning to any 64K page.
Optional SC/DOS file disk operating system.

SC32 Forth Microprocessor

32-bit CMOS microprocessor, 34,000 transistors.
One-clock cycle instruction execution.
Non-multiplexed 32-bit address bus and data bus.
16 gigabyte non-segmented data space.
2 gigabyte non-segmented code space.
8 or 10 megahertz full-static operation.
Stack depths limited only by available memory.
Interrupt and interrupt acknowledge lines.
Bus request and bus grant lines with on-chip tristate.
Wait state line for slow memory and I/O devices.
85-pin PGA package.

RTX 2000 Forth Microprocessor

16-bit CMOS microprocessor in 84-pin PGA package.
1-cycle 16x16 parallel multiplier.
14-prioritized interrupts, one NMI.
Two 256-word stacks. Three 16-bit timer/counters.
8-channel multiplexed 16-bit I/O bus.

NC4016 Forth Microprocessor

16-bit, 4MHz CMOS microprocessor in 121-pin PGA.

Ideal for embedded real-time control, high-speed data acquisition and reduction, image or signal processing, or computation-intense applications. For additional information, please contact us at:

SILICON COMPOSERS INC 208 California Avenue, Palo Alto, CA 94306 (415) 322-8763

FORML '89

In Which We Meet in the Woods and Roo Gets Held Up...

BY PETER MIDNIGHT - SAN LEANDRO, CALIFORNIA

(...the article, that is, not the holdup.)

Once again this past Thanksgiving at Asilomar, some of the foremost Forthers from around the world gathered for the Eleventh FORML Conference. This was the tenth annual FORML Conference to be held at California's beautiful Asilomar Conference Center on the Monterey Peninsula, just a short walk from the world famous Pacific Ocean. However, this conference was officially the eleventh because there was once another one someplace else and there never was a zeroth.

As always, this year's FORML conference had a theme. But the theme is really just a formality to help get the ball rolling. The theme this year was Object-Oriented Programming. And this topic did come up, from time to time. But the real purpose of the FORML Conference is to bring together a diverse group of serious Forth users and to promote the propagation among them of enthusiasm, ideas, and useful information.

The Woods

The first thing you see when you arrive at the conference center is that you are not in Kansas anymore. You are in the woods. Asilomar is like a group-event-oriented resort. The flagpole in the middle is surrounded by the Dining Hall, the lodge-like Administration Building, the Chapel, and the Barbecue Pit. From this area, paved footpaths, going slightly uphill in both directions, spread off through the trees to interconnect a variety of housing and meeting facilities of various sizes. The buildings have names like Spindrift, Manzanita, and Forest Lodge. Inside are wood-burning fireplaces, viewgraph projectors, and vats of less-than-ideal coffee. Outside, you're sure to see a few deer.

Participation in this event begins as soon as you arrive at Asilomar. The first stop is the Administration Building for registration. But before you even get there from your car, you begin to encounter other conference attendees. You meet a few more as you stop to pick up your notebook and meal ticket. By the time you sit down for lunch, you are among the people you have come here to see. Without leaving sea level, you have reached the mountain top, many thousands of feats above C level. For the next 48 hours, you can put the rest of your life on hold.

Formal FORML

After lunch, the conference begins in earnest, with our first session held in the Chapel. Here we learn that Terri Sutton, this year's chairperson, is unable to attend due to illness and that John Hall will cover for her during the conference itself. We also get the first of many handouts to be added to our already full notebooks. This is the first time we all get to open and close our three-ring binders together while someone is trying to speak.

After a few more opening remarks, we get down to business. Here's how that works. The papers that were submitted by the latest deadline were then organized into about half a dozen subject areas. The groupings this year are looking into Forth; comparing Forth; measurements and mathematics; objects and graphics; matching, control flow, and F-PC; ANSI report and assembler innovations; and the future. These groups of papers become most of the sessions to be held at the conference and later become the chapters in the published proceedings. The conference chair, John as Terri, talks someone into chairing each session. The session chair introduces the author of each paper and keeps track of time. Each author gets to address the conference for up to 15 minutes. They usually start by talking about whatever their paper is about. Then they might take a few questions, which may include comments. This interplay dissolves into a discussion that nobody wants to end until the session chair says we have to move on.

From the very beginning, this year, we find ourselves representing opposing points of view on a variety of issues. Some of us believe that long, descriptive word names make code more readable, while others of us believe that long names are as tedious to read as they are to write and leave less room on the printed page for reflecting logical structure or flow in the layout of the text. Some of us fear that the need for the efficiency of Forth will diminish with the cost of hardware, while others of us are hopeful that the demand for that efficiency will increase with the proliferation of uses for ever smaller and more numerous machines. And some of us appreciate the power of automated applications that don't bother us with the details of what all they have to do to achieve their results, while others of us are more concerned about the stifling effect this style of program has on the users' ability to understand and effectively manage the operation of their own computers. Conveniently, this latter conflict foreshadows the subject of Object-Oriented Programming, which is sure to come up sooner or later.

Continuity

The conference proceeds at a steady pace, even between sessions. The break between the afternoon sessions is half an hour long, but you spend that time waiting in line for your room key because this is your only opportunity to check in. Then you make your way up to the Firelight Forum, where the remainder of the conference will be held this year. The dinner break is longer, but by the time you have been served your dessert—if you care to wait for it—it's about time to get started back up the hill for the evening session.

Somewhere, your stomach is trying to make sense of what you've just eaten. At the same time, your brain is digesting a tasty selection of more abstract subjects. In this session, we are treated to a discussion of the application of mathematics to data types other than numbers. We also receive a rare explanation of what a CRC really is and how it can be practical to compute one.

As the evening session draws to a close, more or less on schedule, the wine and cheese are already being served at the back of the room. With more than six hours of presentations already under our belts, we have plenty of food for thought and discussion, as well. In addition, this wine and cheese party is where we are joined by the guests that have accompanied some of us to the conference. They have been off someplace, enjoying non-computing activities hosted by Min Moore.

Officially, the party ends at midnight. All meeting facilities are supposed to close at this time. Those of us who don't give up that easily usually end up packed into someone's bedroom with the last of the wine. This is not an optimum situation for the neighbors and can be even worse for the people whose room has been overrun. This year, we find a beautiful lounge in one of the buildings allocated to attendees of our conference. This lounge is not locked and shares no walls with any bedrooms. Whether by luck or chicanery, this is a great improvement over past years. There is no telling how late into the night the last few diehards among us are still in conference in that room.

Breakfast comes at an hour some programmers have never seen. Fortunately, coffee and doughnuts will be available during the morning break. It's only the second day, and some of us are already suffering from sleep deprivation. But at nine o'clock, ready or not, the presentations begin anew.

Communication

The theme of the conference this year could well have been communication. Whether in screens or in sequential files, the source code we write is designed to communicate both with a machine and with whatever person or persons will need to deal with it in the future. All of our discussions of coding and commenting style address this problem. Even the occasional mention of Object-Oriented Programming refers to a method of selecting the information about each element of a program that will be communicated to a programmer or a user. Other problems discussed at the conference include communication of our understanding of the use of Forth to new and potential users and communication of the practical advantages of Forth to the engineers and managers who are the potential market for our skills and services. And, of course, the object towards which the FORML Conference is most directly oriented is an opportunity for each of us to communicate in person with some of our peers.

For some of us, standing at a microphone and speaking to about 85 of the most knowledgeable Forth programmers in the world is like chatting with friends over a few drinks. For others of us, it is more like having to improvise a song *a cappella* on live network television in the nude! However, we are usually too interested in the content of what other Forth programmers have to say to be concerned about the style of their presentations, as long as they don't pass out.

The style of our graphics, on the other hand, is interesting to observe. In the absence of large video monitors, most presenters make at least some use of the viewgraph projector. Some just use it like a blackboard, while others have prepared slides to illustrate their work with varying degrees of polish. One presenter this year has used a pen plotter to render his viewgraph slides in color. And once, at a previous FORML Conference, we even saw movies.

The proceedings, when they are published, will surely imply that this reporter is the only remaining Forth programmer without a laser printer. But even if that were true, the hardware we use should not be as significant to us, as programmers, as is the software we use. What portion of this printed material do you suppose was developed and formatted through the righteous invocation of Forth, as opposed to that unclean portion which was deep-fat fried by some autocratic word processor written in one of the more fascistic languages, approved by the Ministry of User Friendliness and sold, along with ozone-eating chemicals and used cars, by megalomaniacs with thinly veiled Mafia connections? We shall not dignify this sorry state of affairs with any further speculation.

Informal FORML

Over the years, several different types of sessions have been mixed in with the individual, oral presentation sessions at the FORML Conference. On Saturday afternoon this year, we have working groups. At the beginning of such a session, several areas in the room are assigned to specific areas of interest. Discussions then take place simultaneously in all of these areas, each involving whomever of us find them the most interesting. Those of us not blessed with a one-track mind tend to wander around a bit during the working group session.

The evening session provides another alternative form, impromptu talks. By this time, a great many thoughts have been churned up by the presentations, the working groups, and the many informal conversations. There are also people among us with ideas to present who might have submitted a paper, but didn't. The impromptu talks session is an opportunity for those thoughts and ideas to be presented. And it blends smoothly into another wine and cheese party on Saturday night.

The second night of the conference passes much the same as the first. The same lounge is found unsecured, although it may be empty a little earlier this morning. And again, breakfast is strongly rumored to have taken place as scheduled. If you make it to breakfast and your roommate doesn't, or vice versa, you might go through the entire conference without ever finding out who your roommate is. It might even be the person you are sitting right next to when the final day of the conference begins.

Computers

Computers, themselves, have played an interesting role in the FORML Conference over the years. A decade ago, as you may recall, setting up a computer system at a conference was something of a project in itself. The first computer this reporter saw demonstrated at a FORML Conference was as offbeat as the undertaking of bringing it there. Instead of a cooling fan, it had

(Continued on page 31.)

The Results of Our

CONTEST OF SORTS

DENNIS RUFFER - SYSOP, GEnie FORTH ROUNDTABLE

The results are in, and we have a winner. Although we only had three entries, they came from three distinct corners of the world. From Australia, David Doupe made his submission through the Usenet extension of our Virtual Forth Network. From Denmark, Henning Hansen sent his entry direct to the FIG offices. From our own back yard in Santa Cruz, California, Dwight K. Elvey made a remarkable winning entry. In fact, Dwight made two entries, one written for speed, the other written to minimize memory usage. I have included both, since they both are more than two times faster (on average) than their nearest competitor. I have also included code from Wil Baden, who has taken the time to further discuss [see below] the history of Dwight's method.

The requirements for a sorting algorithm are varied, and should be determined more by the application than simply by the result of an abstract benchmark like we have used here. However, I might mention a few notes about our results. Although Dwight's DVD&KNKR is by far the fastest, both it and David's MERGE produce consistent results no matter how the input data is ordered. It should also be noted that they both take significantly more memory than Henning's FIGSORT. In fact, Dwight's entry is so unique that my tests do not adequately allow for his method. So the saying goes about lies and benchmarks! However, rules are rules, and Dwight deserves the credit for winning under the rules of this contest.

Therefore, with no further adieu, here are the entries.

First Place
DVD&KNKR.ARC
Author: Dwight K. Elvey
Santa Cruz, California

"This sorting algorithm can be used as a general-purpose sort. The precedence or significance of the sort can be changed on the fly. This allows things like ignoring upper and lower case, putting numbers after letters, or whatever, with almost no penalty in execution time. The time it takes to complete the sort is proportional to the number of items, and doesn't grow at some exponential rate like most of the more common sorts. This sort is similar to a Hollerith card sorter.

"This sort is also similar to one I wrote for a friend who was doing mail sorting with Quick sort and was disappointed with the speed. When he saw the speed comparison, he said I couldn't be sorting so fast and there must be a mistake! The one problem is that this sort doesn't use a *compare* function, but the rules require that I use one; so I will use it once to waste time. Since the purpose of this contest is to find the best sort, I feel this is within the spirit of things.

"The main disadvantage of this sort is that it does require more memory. This isn't normally a problem, since the data to be sorted is normally on disk and memory is cheap. For sorting strings, one could sort the links first then reorder the data, but that requires more memory."

The Original Sort—
A Commentary by Wil Baden

Congratulations to Dwight K. Elvey for his implementation of Radix sort, a.k.a. Digit sort, Pocket sort, Basket sort. (See Knuth's, *The Art of Computer Programming*, volume three.)

Given a thousand perfectly random values, Insertion sort will make about 250,000 comparisons, and Quick sort about 13,000 comparisons. On a two-byte field, the Basket sort will make exactly 2000 examinations. A comparison involves fields from two records, but an examination involves just one byte of one record. Thus, examinations should be more than twice as fast as comparisons. So a Basket sort really flies here.

The Basket sort was used in the 1920s by IBM before it was called IBM. International Tabulating Company (or Corporation) was its name then.

I believe the Basket sort is the origin of the word "sort" in its computing sense. The original meaning of sort is "classify." Sorting laundry, you put white stuff into one pile, or basket, dark colors into another, other colors into another, delicate hand-washable stuff into another, etc. A similar procedure was done with punched cards.

First, the penny column was taken to sort (i.e., classify) cards into a batch for each digit, then the dime column, the dollar column, the sawbuck column, the yard column, the grand column.

Since the Basket sort is the original sort, and is so fast, why isn't it better known?

The major disadvantage is not memory, but the number of passes needed. In commercial applications, it is common to sort on 30 or more columns. This would take 30 or more passes in a simple-minded Basket sort, no matter how many records there were. Using Quicksort or Heapsort would take log2(N) passes, where N is the number of records.

With large memories, basket sorting may make a comeback. A file to be ordered by nine-digit social security numbers can be sorted in three passes by taking the number in "base 1000."

[Wil Baden's code follows the contestants' entries.]

DVD&KNKR.ARC
Speed-optimized version's test results

Test	*Dict*	*RAM*	*Fetches*	*Stores*	*Compares*	*Time*	*Score*	*Max.*	*Avg.*
RAMP	7052	37	1024	1024	1	0.71	6.91	6.98	6.84
SLOPE	7052	35	1024	1024	1	0.71	6.91	6.98	6.84
WILD	7052	32	1024	1024	1	0.66	6.56	6.98	6.84
SHUFFLE	7052	31	1024	1024	1	0.72	6.98	6.98	6.84
BYTE	7052	31	1024	1024	1	0.71	6.91	6.98	6.84
FLAT	7052	36	1024	1024	1	0.71	6.91	6.98	6.84
CHECKER	7052	31	1024	1024	1	0.72	6.98	6.98	6.84
HUMP	7052	37	1024	1024	1	0.66	6.56	6.98	6.84

DVD&KNKR.ARC
Memory-optimized version's test results

Test	*Dict*	*RAM*	*Fetches*	*Stores*	*Compares*	*Time*	*Score*	*Max.*	*Avg.*
RAMP	4952	32	2048	2048	1	0.88	8.28	8.28	8.13
SLOPE	4952	36	2048	2048	1	0.88	8.28	8.28	8.13
WILD	4952	31	2048	2048	1	0.88	8.28	8.28	8.13
SHUFFLE	4952	36	2048	2048	1	0.88	8.28	8.28	8.13
BYTE	4952	36	2048	2048	1	0.82	7.99	8.28	8.13
FLAT	4952	36	2048	2048	1	0.82	7.99	8.28	8.13

FIGSORT.ARC
Test results

Test	*Dict*	*RAM*	*Fetches*	*Stores*	*Compares*	*Time*	*Score*	*Max.*	*Avg.*
RAMP	676	84	7255	0	7062	1.92	15.40	32.03	19.91
SLOPE	676	84	7280	1024	7084	2.14	16.60	32.03	19.87
WILD	676	68	12933	4717	12000	3.95	31.81	32.12	20.03
SHUFFLE	676	76	12879	4662	11930	3.95	31.68	32.12	20.19
BYTE	676	76	11451	4554	10659	3.57	28.65	32.12	20.28
FLAT	676	33	1028	0	1026	0.28	2.19	32.12	20.05
CHECKER	676	72	3817	1450	3784	1.21	9.71	32.12	19.92
HUMP	676	80	10440	4308	9796	3.30	26.39	32.12	20.00

MERGE.ARC
Test results

Test	*Dict*	*RAM*	*Fetches*	*Stores*	*Compares*	*Time*	*Score*	*Max.*	*Avg.*
RAMP	2558	58	14848	4608	5120	3.08	31.86	59.70	48.69
SLOPE	2558	64	20480	10240	5120	4.34	46.11	59.70	48.67
WILD	2558	63	27275	9337	8969	5.49	58.56	59.70	48.80
SHUFFLE	2558	66	27277	9349	8964	5.50	58.61	59.70	48.93
BYTE	2558	65	27220	9292	8964	5.49	58.47	59.70	49.04
FLAT	2558	54	14848	4608	5120	3.07	31.83	59.70	48.82
CHECKER	2558	63	21760	6912	7424	4.39	46.48	59.70	48.80
HUMP	2558	62	27187	9305	8941	5.49	58.40	59.70	48.92

DVD&KNKR.ARC

```
: ARRY ( 16 bit array maker )
  ( Size-in-items ) CREATE 2* ALLOT
  ( Index - Addr ) DOES> SWAP 2* + ;

256 CONSTANT SIZEROOT ( don't change for more data )
```

(Sort code continues.)

(Continued from page 28.)

a chimney. There was a little sign on the chimney that said, "Thank you for not smoking!" And the software it ran was an equally creative expression of Forth, with no terminal input buffer and all the names stored backwards in the dictionary.

Some years later, a few Model 100s started showing up. That is when we discovered how distracting the sound of typing can be during an oral presentation. A few people still take notes on computers at the conference. But they usually sit in the back, type very gently and quietly, and for the most part avoid typing when someone is speaking.

In recent years, when Forth engines began to hit the scene, we started getting product announcements and flyers. Several machines appeared at the conference to show off their speed. A typical demonstration would consist of a box or two of electronics and a 286 workstation acting as a dumb terminal. When told to go, the entire system would sit quietly for a few moments and then print out, "Look how fast I did that!" Each such system was backed up by an enthusiastic entrepreneur to tell you what it had just done. By listening carefully, you could tell that this was pretty hot stuff you were not quite seeing.

This year, we see very few computers. Even the assortment of laptop machines that is here is mostly not in evidence except during the breaks. One handmade RTX 2000 computer, somewhat smaller than a software package, is held up with pride before the conference. Have we come full circle yet? Perhaps not. That machine is for sale for $14,000 in moderate quantities. And it doesn't even have a chimney!

Conclusion

At the close of the FORML Conference, a bottle of wine is awarded to each of several participants whose presentations have been judged outstanding in one way or another. And two or three attendees are asked to say a few words about their experience of the conference. Some of the observations offered this year are that there is an awful lot of talking at the conference and probably even some listening, that typical Forth programmers here are older than typical Forth programmers in Finland, and that the presentations at the conference are

(Continued on next page.)

(Sort code continued.)

```
SIZEROOT ARRY LINKROOT   ( This is the divide part )
ITEMS ARRY DATALINK  ( Links to data )
ITEMS ARRY DATATMP1  ( More space )
ITEMS ARRY DATATMP2  ( More space )
VARIABLE DATDIV ( used for fast 256 / to separate bytes )
DATDIV 1+ CONSTANT DAT/256  ( Remove 1+ for machines like 68K )

 : INITLINK ( init links )
    ( Links need a marker for end of chain. )
    ( Since 0 is a valid pointer I'll use -1 )
    [ 0 LINKROOT ] LITERAL
    [ SIZEROOT 2* ] LITERAL -1 FILL ;                  ( 14:38

 : REORDER1 ( InsertPointer RLinkTo RLinkFrom - InsertPointer' )
  DO  I ( By modifying the root link pointer at )
    ( this point one could change the sort order )
    ( like letters, numbers and then puncts, )
    ( Through a translation table using ARRY )
   LINKROOT @ ( First Link in Chain )
   BEGIN SWAP OVER 1+   ( end of link marked by -1 )
   WHILE OVER DATATMP1 @   ( Fetch data to restore )
    OVER DATATMP2 ! ( Reorder Data )
    1+ ( Incr INSERT Pointer ) SWAP
    DATALINK @ ( Follow link ) REPEAT
   SWAP DROP ( Save Insert Pointer )
  LOOP ;

 : LSB ( Items - , Sort the LSB's first )
  INITLINK ( Clear root links )
  0 DO ( This builds an ordered link list of items )
    I S@ ( Order isn't important on first pass like in MSB )
    DUP I DATATMP1 ! ( Fetch and save data )
    255 AND ( 256 MOD separate LSB byte )
    LINKROOT DUP @ ( Fetch old link and put new )
    I DATALINK ! ( Build links )  I SWAP ! ( New in LINKROOT )
  LOOP   ( Lsb links now made so put data back )
  0 256 0 REORDER1 ( reorder entire array ) DROP ;

 : REORDER2 ( InsertPointer RLinkTo RLinkFrom - InsertPointer' )
  DO  I ( By modifying the root link pointer at )
    ( this point one could change the sort order )
    ( like letters, numbers and then puncts, )
    ( Through a translation table using ARRY )
   LINKROOT @ ( First Link in Chain )
   BEGIN SWAP OVER 1+   ( end of link marked by -1 )
   WHILE OVER DATATMP2 @   ( Fetch data to restore )
    OVER S! ( Reorder Data )
    1+ ( Incr INSERT Pointer ) SWAP
    DATALINK @ ( Follow link ) REPEAT
   SWAP DROP ( Save Insert Pointer )
  LOOP ;

   : MSB ( Items - , Sort the MSB's next )
    INITLINK ( Clears root links ) 1- ( Last item )  0 SWAP
    DO ( This builds an ordered link list of items )
      I DATATMP2 @ ( Reverse order not to undo LSB's work )
      DATDIV ! DAT/256 C@ ( does 256 / unsigned )
      LINKROOT DUP @ ( Fetch old link and put new )
      I DATALINK ! ( Build links) I SWAP ! ( New in LINKROOT)
    -1 +LOOP    ( MSB links now made so put data back )
    0 256 128 REORDER2 ( Do negative values first )
      128 0 REORDER2 ( now positive numbers ) DROP ;

   : DVD&KNKR ( #Items - ) ( Divide and Conquer )
      DUP LSB MSB 0.0 COMPARE DROP ;
```

"This sort should score about 7.70 average on the test and have a very consistent time. I also have one that uses about 70% less dictionary but does twice the fetches and stores. It would have a score of about 8.10 average on this test, but dictionary is less important than fetches and stores. It seems that most of the variations I came up with had similar times, not varying by more than ten percent. "I have included the more memory efficient version on blocks eight through eleven, since it is better code and more useful but doesn't score as well on this test."

(Sort code continues.)

(Sort code continued.)

```
: ARRY ( 16 bit array maker )
  ( Size-in-items ) CREATE 2* ALLOT
  ( Index - Addr ) DOES> SWAP 2* + ;

256 CONSTANT SIZEROOT ( don't change for more data )
SIZEROOT ARRY LINKROOT  ( This is the divide part )
ITEMS ARRY DATALINK ( Links to data )
ITEMS ARRY DATATMP  ( More space )
VARIABLE DATDIV ( used for fast 256 / to separate bytes )
DATDIV 1+ CONSTANT DAT/256 ( Remove 1+ for machines like 68K )

: INITLINK ( init links )
   ( Links need a marker for end of chain. )
   ( Since 0 is a valid pointer I'll use -1 )
   [ 0 LINKROOT ] LITERAL
   [ SIZEROOT 2* ] LITERAL -1 FILL ;

: REORDER ( InsertPointer RLinkTo RLinkFrom - InsertPointer' )
 DO  I ( By modifying the root link pointer at )
   ( this point one could change the sort order )
   ( like letters, numbers and then puncts, )
   ( Through a translation table using ARRY )
  LINKROOT @ ( First Link in Chain )
  BEGIN SWAP OVER 1+  ( end of link marked by -1 )
  WHILE OVER DATATMP @  ( Fetch data to restore )
   OVER S! ( Reorder Data )
   1+ ( Incr INSERT Pointer ) SWAP
   DATALINK @ ( Follow link ) REPEAT
  SWAP DROP ( Save Insert Pointer )
 LOOP ;

: LSB ( Items - , Sort the LSB's first )
 INITLINK ( Clear root links )
 0 DO ( This builds an ordered link list of items )
   I S@ ( Order isn't important on first pass like in MSB )
   DUP I DATATMP ! ( Fetch and save data )
   255 AND ( 256 MOD separate LSB byte )
   LINKROOT DUP @ ( Fetch old link and put new )
   I DATALINK ! ( Build links )  I SWAP ! ( New in LINKROOT )
 LOOP   ( Lsb links now made so put data back )
 0 256 0 REORDER ( reorder entire array ) DROP ;

: MSB ( Items - , Sort the MSB's next )
 INITLINK ( Clears root links ) DUP 1- ( Last item ) SWAP
 0 DO ( This builds an ordered link list of items )
   DUP I - S@ ( Reverse order so as not to undo LSB's work )
   DUP I DATATMP ! ( fetch and save data )
   DATDIV ! DAT/256 C@ ( does 256 / unsigned )
   LINKROOT DUP @ ( Fetch old link and put new )

   I DATALINK ! ( Build links )  I SWAP ! ( New in LINKROOT )
 LOOP DROP ( MSB links now made so put data back )
 0 256 128 REORDER ( Do negative values first )
   128 0 REORDER ( now positive numbers ) DROP ;

: DVD&KNKR ( #Items - ) ( Divide and Conquer )
   DUP LSB MSB 0.0 COMPARE DROP ;
```

Second Place
FIGSORT.ARC
Author: Henning Hansen
Lyngby, Denmark

```
VARIABLE LO VARIABLE MED VARIABLE HI VARIABLE ?EQ VARIABLE ?FIN

\ place high item after selecting all smaller items
: SELECT-SMALLER ( high low -- high low )
   HI @ >R OVER 1- SWAP
   BEGIN 2DUP < NOT
   WHILE R@ 2 PICK S@ DUP HI !
     COMPARE 0< IF SWAP 1- SWAP ELSE RECURSE THEN
```

(Sort code continues.)

(FORML, continued.)

just an excuse for us to be here during the breaks, when the real interaction takes place. It is also noted that there has been almost no mention of Pooh Forth this year, although that $14,000 computer we saw earlier did bear the designation ROO.

After lunch, with most of your goodbyes said, you need a segue back to the real world. It has been 48 hours since the conference began, just about the length of time we humans seem to need in order to achieve saturation. This is the opportunity some of us take to head off across the dunes to the beach. This used to be a tedious trek and somewhat harmful to the dunes, as well. But now a boardwalk leads directly from the Barbecue Pit to the Coast Road and the beach.

The beach affects us each in different ways. Some of us use this time to stroll on the sand while we mentally emerge. Others tend to stare out to sea, seeking to gain some perspective and letting the ocean put us in our place. Still others just go right on talking Forth or rejoin the wives or other guests we may have brought along to the conference. Here is where you look back upon the conference, look forward to the coming year, and know where you will be next Thanksgiving.

Peter Midnight was an audio and video technician until he got involved with computers in 1977. He has attended all ten FORML conferences at Asilomar and, since 1984, has been an engineering consultant specializing in embedded systems.

(Continued from page 25.)

ALIGN adjusts the dictionary pointer to the first aligned address greater than or equal to its current address. An aligned address is suitable for storing or fetching bytes, cells, cell pairs, or double-precision numbers.

After initializing the table, we would also like to read values from it. For example, assume we want to fetch the first cell, X, from the table. ATABLE BYTE+ gives the address of the first thing after the byte. However, this may not be the address of X since we aligned the dictionary pointer between the C, and the ,. The portable way to get the address of X is:

```
ATABLE BYTE+ REALIGN
```

REALIGN adjusts the address on top of the stack to the first aligned address greater than or equal to its current value.

Example

Let's pull several of the techniques just described into a single example. Let's design a machine-independent facility for building Pascal/C-style record structures. Listing One shows the syntax for declaring a structure, creating an instance of a structure, and accessing its members. FOO is a structure consisting of three parts: a three-byte member, a single-cell member, and a single-byte member. The FOOBAR structure consists of two cells and a FOO substructure. The '.' in the name of a member is convention to make C and Pascal programmers feel at home.

The implementation also appears in Listing One. The order in which the members appear in a structure declaration is roughly reflected in the memory layout of a structure instance—roughly, because the structure compiler may place padding between members to avoid alignment problems. Each member defining word adjusts a template address by an appropriate size. The guts of the compiler, AUS:, adjusts the template address by a given number of address units. So, BYTES: uses BYTES to compute the number of address units needed by its member and calls AUS: to allocate it. CELLS: works similarly but it aligns the template address first.

The record-structure implementation has a number of nice features. The ANS Forth operators BYTES, CELLS, and REALIGN handily hide hardware details. The correct alignment of structure members is handled automatically by the structure compiler. Observe that scaling and alignment are done at compile time. The structure is itself a word that returns the size of the structure in address units. A way of finding the size of a structure is essential, since it will vary from system to system. The size can be used with ALLOT to allocate a structure instance or with MOVE to copy a structure.

Summary

This article has described how to use data and memory portably in ANS Forth. Of course, there are other aspects of portability. For example, different computers

(Sort code continued.)

```
    REPEAT SWAP DROP 2DUP >
    IF DUP S@ DUP HI ! 2 PICK S! R> OVER S! ELSE R> DROP THEN
    1+ ;

 \ sort few items using selections
 : SORT-A-FEW ( high low -- )
    OVER S@ HI ! BEGIN 2DUP > WHILE SELECT-SMALLER REPEAT
    2DROP ;

 \ order three items
 : ORDER-THREE ( high med low -- )
   ROT DUP >R S@ ROT DUP >R S@ ROT DUP >R S@
   2DUP COMPARE 0< 2OVER COMPARE 0<
   2DUP OR NOT
   IF 2DROP R> R> R> 2DROP 2DROP 2DROP
   ELSE OVER AND
     IF DROP ROT R> S! R> DROP R> S! DROP
     ELSE
       IF SWAP R> S! 2DUP COMPARE 0< IF SWAP THEN R> S! R> S!
       ELSE ROT 2DUP COMPARE 0< IF SWAP THEN R> S! R> S! R> S!
       THEN
     THEN
   THEN ;

 : ON  ( addr -- )  -1 SWAP ! ;
 : OFF ( addr -- )   0 SWAP ! ;

 \ partition low and high ends of interval
 : PARTITION ( high low med — h.high h.low l.high l.low )
    S@ MED ! ?FIN ON 2DUP SWAP
    BEGIN ?EQ ON SWAP
      BEGIN 1+ DUP S@ DUP LO ! MED @ COMPARE

        DUP 0<> IF ?EQ OFF THEN 0< NOT
      UNTIL SWAP
      BEGIN 1- MED @ OVER S@ DUP HI ! COMPARE
        DUP 0<> IF ?EQ OFF THEN 0< NOT
      UNTIL ?EQ @ NOT IF ?FIN OFF THEN 2DUP <
    WHILE ?EQ @ NOT IF LO @ OVER S! HI @ 2 PICK S! THEN
    REPEAT  2DUP = IF 1+ SWAP 1- THEN
    ?FIN @ IF 2DROP 2DUP THEN ROT ;

 15 CONSTANT MANY

 \ sort from low to high using medium-of-three partition
 : SORT-THEM-ALL ( high low -- )
    2DUP - MANY <
    IF SORT-A-FEW
    ELSE
      2DUP 2DUP + 2/ SWAP ORDER-THREE
      2DUP + 2/ PARTITION
 \    2OVER 2OVER - + < IF 2SWAP THEN   \ smallest part first
      2DUP > IF RECURSE ELSE 2DROP THEN
      2DUP > IF RECURSE ELSE 2DROP THEN
    THEN ;

 \ use FIGSORT to sort the items numbered 0 to n-1
 : FIGSORT   ( n -- ) 1- 0 SORT-THEM-ALL ;
```

Third Place
MERGE.ARC
Author: David Doupe
Woollahra, NSW, Australia
Though Merge sort is not spectacularly fast, it does have the advantage that it is a *stable* sort; i.e., sorting on one key (field) does not disturb previous sorting operations done on other keys. Also, it is O(n log n). The contest requirements result in reduced efficiency for this implementation.

```
 CREATE DATA2  ( — a P:Array for emerging lists)
                         ITEMS CELLS ALLOT

 VARIABLE N-DATA2  ( — a P:Count for newly emerging list )
```

(Sort code continues.)

(Sort code continued.)

```
: S2! ( n - )  N-DATA2 @  CELLS DATA2 + !  1 N-DATA2 +!
               STORES !USE ;

VARIABLE A1  VARIABLE A2
VARIABLE N1  VARIABLE N2
 ( pointers & counters to the two current sublists )

VARIABLE PL1 VARIABLE PL2
 ( pointers to latest item# selected in each sublist )

: A1@   ( - n )  A1 @ S@ ;
: A2@   ( - n )  A2 @ S@ ;

: INCA1 ( - f )  1 A1 +!  1 PL1 +!  PL1 @ N1 @ > ;
  ( Having chosen from sublist1, increment A1 and PL1, then
                       test if sublist1 is exhausted )
: INCA2 ( - f )  1 A2 +!  1 PL2 +!  PL2 @ N2 @ > ;
  ( Having chosen from sublist2, increment A2 and PL2, then
                       test if sublist2 is exhausted )

: <MRG>   ( - )
     BEGIN  A2@  A1@  COMPARE  -1 =
      IF   A2@  S2! INCA2
          IF  N1 @ PL1 @ - 1+ 0
               DO   A1@ S2!  1 A1 +!
               LOOP  EXIT
           THEN
       ELSE  A1@ S2!  INCA1 IF EXIT THEN
       THEN
     AGAIN  ;

: SET-UP-MRG ( s1 n1 s2 n2 -  s1 )
        N2 ! A2 ! N1 ! DUP A1 !
        1 PL1 ! 1 PL2 !  0 N-DATA2 !  ;

: TO>FROM-MOVE ( s1 - s1 ) DATA2   OVER  CELLS DATA + ( s1 s d )
                       N-DATA2 @ CELLS   ( s1 s d n ) MOVE ;
 ( This does not increment the counted fetches and stores )

: MRG ( s1 n1 s2 n2 - s1 n1+n2 )
        SET-UP-MRG  <MRG>  TO>FROM-MOVE
        ( s1 )    N1 @ N2 @ +  ;

: RA-SPLIT ( s1 n - s1 n1 s2 n2 )    ( n1<= n2 )
        2 /MOD SWAP OVER + >R 2DUP + R> ;

: 2CHKSWAP   ( n - )   ( compare an adjacent pair & swap if nec)
              DUP S@ OVER 1+ S@ SWAP
              COMPARE  -1 =
              IF    DUP 1+ EXCHANGE EXIT
              ELSE  DROP
              THEN   ;
( This could be done quicker with 2@ and 2! since items are
                        always adjacent )

: MSOCLIP ( s1 n1 - s1 n1 P:Deal with recursion exit conditions)
      DUP 2 =  IF  OVER  2CHKSWAP R> DROP EXIT  THEN
      DUP 2 <  IF  R> DROP EXIT THEN
    ;

        (  THIS IS THE MAIN ALGORITHM  )
: <MSORT>   ( s1 n1 - s1 n1 )
        MSOCLIP
        RA-SPLIT
        RECURSE  2SWAP  RECURSE  2SWAP
        MRG  ;

: MSORT   ( # - ) 0 SWAP
              <MSORT>  2DROP   ;
```

(Sort code continues on page 39.)

represent numbers in different ways and dependence on a particular representation should be avoided. Assumptions about the underlying Forth implementation should also be avoided. During Forth's history, an amazing variety of implementation techniques have been developed. The ANS Forth standard encourages this diversity and consequently restricts the assumptions that a user can make.

There is no such thing as a completely portable program. A programmer should intelligently weigh the tradeoffs of providing portability to specific machines. For example, machines that use sign-magnitude numbers are rare and probably don't deserve much thought. But systems with different cell sizes will certainly be encountered and should be provided for. In general, making a program portable clarifies both the programmer's thinking process and the final program.

This issue also contains "SC32: a 32-Bit Forth Engine" by the same author.

(Editorial, from page 4.)

embedded-systems programming is drawing from the Forth labor pool, reportedly in increasing numbers. (Only heaven knows whether even a sweeping endorsement by industry would crack open the ivory gates of academia at large to Forth coursework, but it would be a great vindication to all the engineering departments happily using Forth in their laboratory classes.)

One last thing...

We have published more pages this year than ever before, and we hope to continue doing so. It is only your membership in the Forth Interest Group that enables us to stay in print. Like public television, we hope you will vote with your checkbook to keep bringing quality Forth techniques and developments to you. Watch for your membership renewal notice, and return it soon. We anticipate an exciting year ahead—for the Forth language, the industry, and FIG—and we will bring the best and most important developments to you in this members' magazine.

—Marlin Ouverson
Editor

BEST OF GENIE

GARY SMITH - LITTLE ROCK, ARKANSAS

News from the GEnie Forth RoundTable—The X3/J14 ANS Forth Technical Committee, charged with the task of writing an ANS Forth Standard, has for some time discontinued official sanction of GEnie (or any other public information service) as repository of comment and feedback to their efforts. This does not mean there is no standards activity on the GEnie Forth RoundTable—few things could be further from the truth. Comments are still being made and they are being monitored. By the time this makes print, we will also have had X3/J14 chair Elizabeth Rather as our guest in conference, her topic being the X3/J14 ANS standards effort. The future of Forth is being debated. Are you denying yourself a voice in this most important event ? Read the following recently posted exchanges and if something strikes a chord of harmony—or disharmony—join in.

* * *

From: Roedy Green
Subj: IEEE floating point

If you specify IEEE binary format, mainframers will simply have to ignore you. The overhead of converting to IEEE in F! would be enormous and silly. If you don't specify IEEE format, the micro people will all use it anyway because that is how the 80387, etc., work. So my vote goes for leaving it out. We might have a document on a data interchange format where IEEE has a big role. This is really outside the realm of the Forth language, though.

To: Roedy Green
From: Jack Brown
Subj: IEEE FP

Since I have been the major proponent for the inclusion of IEEE, you and others may be interested in knowing that my ear has been bent after listening carefully to several Forth people with mainframe and minicomputer backgrounds. I intend to withdraw my proposal to specify IEEE binary format in favour of one which will specify a word to convert a system's "internal floating-point format" to the IEEE binary format in order to promote exchange of floating-point data.

The future of Forth is being debated.

To: Roedy Green
From: Ian Green
Subj: Language of Forth Standard

On standards I can offer some assistance. First I suggest that, regardless of what the bums at any standards committee have to say, one thing I have *never* seen for Forth is an extended (or standard) Bachus-Naur form of syntax definition for each word, etc. For example, in his book *Programming in Modula-2*, Wirth make several omissions regarding the way the language is supposed to work. He did, however (interspersed throughout the text and again in an Appendix), provide the Bachus-Naur formal definition in absolute precise terms. Because I can understand EBNF, I was able to simply refer to the syntax chart when I had a problem making a piece of code compile (now, of course, I do not need the tables, as I am quite proficient in that language).

My biggest stumbling block about Forth, and many other languages for that matter, is the lack of a formal definition. With a formal definition using EBNF, it is possible to design an unambiguous language standard. Now, Modula-2 is only one of many very serviceable languages and I can also program in a variety of others, but I need the EBNF syntax charts if I am to make any headway. That combined with examples based on the syntax.

To: Ian Green
From: Roedy Green
Subj: Language of Forth Standard

Given the simplicity of the grammar of Forth—no precedence, only space and a separator, strict nesting, I don't see the lack of BNF as an important omission, except to welcome people with Wirthian language backgrounds.

To: Ian Green
From: Jerry Shifrin
Subj: Language of Forth Standard

Actually, for reasons which are beyond me, C.H. Ting did do a BNF for Forth. You can find it in his *Forth Notebook* from Offete Enterprises or in the 2/82 issue of *Dr. Dobb's Journal*. As near as I can tell, the BNF description of Forth should be something like that shown in Figure One [page 41].

To: Jerry Shifrin
From: Ian Green
Subj: Language of Forth Standard

Yes, that is the idea I was looking for. That and examples of code relative to the syntax charts. If you or someone has the complete BNF syntax for Forth (I am currently playing with F83). This, I feel, would go a long way towards clarifying the way the language works.

To be more in keeping with EBNF, I suspect that Forth would be defined something like Figure Two. The problem is, though I can write the syntax for a familiar language fairly easily, Forth is not so easily figured out.

To: Ian Green
From: Jerry Shifrin
Subj: Language of ANS Forth

Actually, Ian, the point I was trying to make is that Forth isn't usually willing to sit still long enough for someone to develop a detailed syntax. <grin>

By that, I mean that every Forth token is eligible for redefinition and may alter the syntax of itself and subsequent string. For example, Forth programs may redefine or add `IF` or `DO` control structures; may redefine constants, e.g.:

```
99 CONSTANT 5
\ 5 now means 99!
```

can even redefine existing functions in terms of themselves, e.g.:

```
: DUP
  DUP ." DUPing " .
  CR DUP  ;
  \ Trace uses of DUP
```

and, in fact, can even redefine defining words, e.g.:

```
: CONSTANT
  CREATE !
  DOES> @ DUP .S  ;
: :
  : 1 #DEFS +! ;
```

which redefines colon in terms of itself and adds a little counter.

So, while you could get a BNF description of typical usage, it doesn't really define the language.

From: Roedy Green
Subj: -LEADING -TRAILING

I am concerned about the string-chopping verbs `-TRAILING` and `SKIP`. There are really six different, but related, words. The complete set need not be made part of the standard, but I think they should be consistently named so that it would be easy to add the missing ones. I think `SKIP` is a misleading name because it implies hopping over a unread record.

Here are my proposed names:

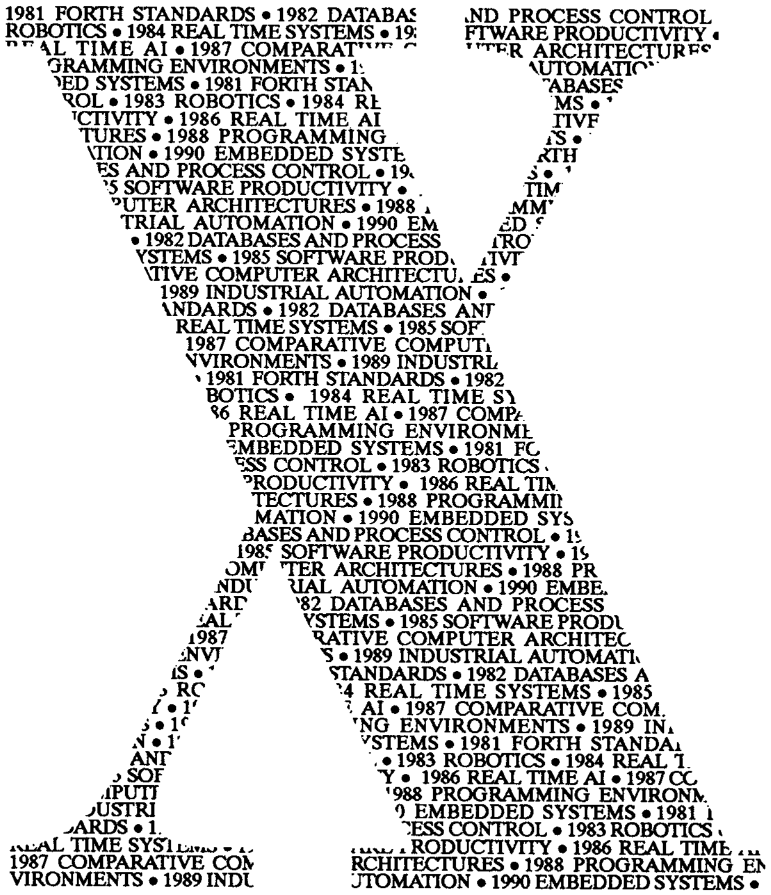
```
-LEADING -LEADING<> -LEADING=
-TRAILING -TRAILING<>
-TRAILING=
```

`-LEADING`
(addr1 +n1 -- addr2 +n2)
Pronounced "dash-leading" or "minus-leading"

Trims any leading blanks from a string. The length may also be zero or one, but not negative. The address and character count

TENTH ANNUAL ROCHESTER FORTH CONFERENCE ON EMBEDDED SYSTEMS

June 12th – 16th, 1990
University of Rochester
Rochester, New York



Call for Papers

There is a call for papers on the use of Forth technology in Embedded Systems. Papers are limited to 5 pages, and abstracts to 100 words. Longer papers will be considered for review in the refereed Journal of Forth Application and Research.

Please send abstracts by **April 15, 1990** and final papers by **May 15, 1990.**

For more information, contact:

Lawrence P. Forsley
Conference Chairman
Institute for Applied Forth Research, Inc.
70 Elmwood Avenue
Rochester, NY 14611
(716)-235-0168 • (716)-328-6426 (FAX)

EMail:	
GEnie	L.Forsley
BIX	LForsley
Delphi	LFORSLEY

WE'RE BOOTING UP!

Proceedings of the 1989 Rochester Forth Conference

6 invited papers and 54 presented papers on all aspects of Forth processors, applications and object oriented technology, including:

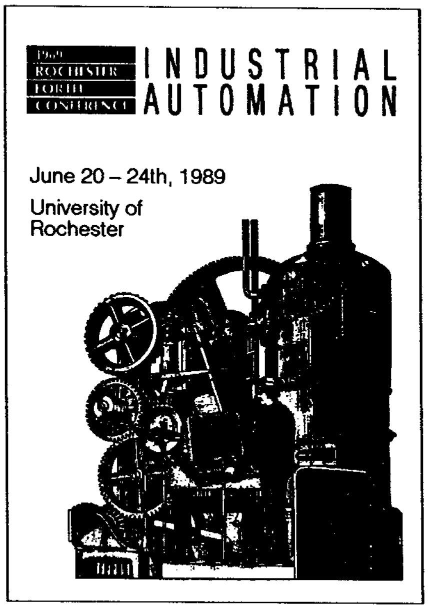
SwissForth, A Development and Simulation Environment for Industrial and Embedded Controllers

Forth-based Control of an Ion Implanter

```
: Cellmate/TOOLBOX
  Hardware/Software
  Workstation/Language DOES>
  Automotive/Aerospace
  Powertrain/Vehicle
  Development/Testing ;
```

Events and Objects: Industrial Control by Hierarchical Decomposition

Breakthrough in Knowledge Management



JFAR Volume 6

Publisher:
Lawrence P. Forsley

Editors:
Dr. S.N. Baranoff, *U.S.S.R Editor*
Leningrad Institute for Informatika

Dr. J. Basile, *Editor-in-Chief*
Long Island University

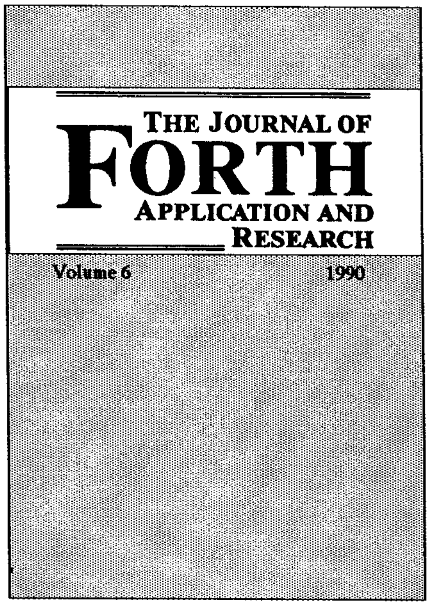
Dr. R. Crawford, *U.K. Editor*
Microprocessor Engineering Ltd.

Dr. M. Kelly, *U.S. Editor*
University of Virginia

Dr. H. Nieuwenhuÿzen, *European Editor*
University of Utrecht, The Netherlands

Dr. N. Solntseff, *Canadian Editor*
McMaster University, Ontario, Can.

Upcoming Papers:
- 32 Bit Forth Processors
- Forth in the U.S.S.R.
- Object Oriented Extensions



NOW'S THE TIME!

Volume VI Subscriptions

	Individual	Corporate
USA	$60.00	$145.00
Canada/Mexico	$65.00	$145.00
Europe/Asia	$75.00	$160.00

1989 Rochester Conference Proceedings
Industrial Automation $25, plus $5 S/H

Send name, full address and phone number. Check or money order in US funds, or, VISA/MC # and exp. date.

To:
Journal of Forth Application and Research
70 Elmwood Avenue
Rochester, NY 14611 USA
(716) 235-0168 • (716) 328-6426 fax
EMail: GEnie ...... L.Forsley
BIX .... LForsley
Delphi ...... LFORSLEY

+n1 of a text string beginning at addr1 is adjusted to exclude leading spaces. If +n1 is zero, then +n2 is also zero. If the entire string consists of spaces, then +n2 is zero. The length n1 must be under 64K and the last character of the string must be covered by Seg of addr. Addr need not be canonical. The original string is unmolested. c.f., `-LEADING= -LEADING<> -TRAILING -TRAILING= -TRAILING<> SCAN SCAN<>`

`-LEADING<>`
(addr1 +n1 char -- addr2 +n2)
Pronounced "minus-leading-not-equal"

Trims any leading characters *that do not match char* from a string. The length may also be zero or one, but not negative. If +n1 is zero, then +n2 is also zero. The length n1 must be under 64K and the last character of the string must be covered by Seg of addr. Addr need not be canonical. The original string is unmolested. c.f., `-LEADING -LEADING= -TRAILING -TRAILING= -TRAILING<> SCAN SCAN<>`

`-LEADING=`
(addr1 +n1 char -- addr2 +n2)
Pronounced "minus-leading-equal"

Trims any leading characters that match char from a string. `-LEADING` is equivalent to `BL - LEADING=`. The length may also be zero or one, but not negative. If +n1 is zero, then +n2 is also zero. The length n1 must be under 64K and the last character of the string must be covered by Seg of addr. Addr need not be canonical. The original string is unmolested. c.f., `-LEADING -LEADING<> -TRAILING -TRAILING= -TRAILING<> SCAN SCAN<>`

`-TRAILING`
(addr +n1 -- addr +n2)
Pronounced "dash-trailing" or "minus-trailing"

Trims any trailing blanks from a string. The length may also be zero or one, but not negative. The character count +n1 of a text string beginning at addr is adjusted to exclude trailing spaces. If +n1 is zero, then +n2 is also zero. If the entire string consists of spaces, then +n2 is zero. The length n1 must be under 64K and the last character of the string must be covered by Seg of addr. Addr need not be canonical. The original string is unmolested.

`-TRAILING<>`

(addr +n1 char -- addr +n2)
Trims any trailing characters *that do not match char* from a string.

-TRAILING=
(addr +n1 char -- addr +n2)
Trims any trailing characters from a string that match char.

To: Roedy Green
From: Jack Brown
Subj: Leaving out words

There used to be a Controlled Reference Wordset, but that has been eliminated and is not likely to be put back in, as it was more like a compromise trash bucket. Words without enough support to get in the standard were thrown into the controlled wordset, supposedly either on their way into or out of the standard next time round. There are still two other wordsets for words that cannot make it into the standard. They are the Reserved wordset for inherently non-portable common usage words like .S, DUMP, etc.; and the Future Directions wordset, for candidates for inclusion in a future standard. TUCK does not fit into either of these wordsets. To reserve the word TUCK, it would have to be included in either the Core wordset or Extended Core wordset.

To: Roedy Green
From: Jack Brown
Subj: What was left out

Thank you for uploading your comments on Basis 10. I will make sure that the editors of it are notified of some of the errors that you have detected. You have made some excellent points; however, if you feel very strongly that certain things should be changed, you should consider making a formal proposal. Forms for doing this are available for downloading. Look for the file ANSITPF.ZIP

Your comments are very likely to cause other members of the ANSI Technical Committee to generate proposals to fix and clarify problems that you have detected. But other ideas, like your -TRAILING and related words, will probably require a proposal generated by yourself to make it to the table for discussion.

I will be looking carefully at your comments when I am preparing my proposals for the January meeting.

From: Zafar Essak
Subj: Basis 10 feedback

Well here goes, more feedback from another BC Forth enthusiast. Having spent an evening sitting around with a few others and discussing some of the concerns raised by a reading of Basis, the first realization is that others can come up with some pretty good justifications for their positions, at least enough to justify having to place definitions in my "Prelude" to accommodate them.

First, I too share Robert Berkey's wish that the FOR ... NEXT looping construct used a word other than NEXT, which seems to be at the heart of the Forth inner interpreter, at least conceptually.

And then...

```
7.0020  "
( -- adr,u)
"quote"
```

I don't know the complete history of this word, but feel strongly that if it returned the address of the count it would be more useful. I realize this will break existing code, but since this word has not been included in a previous standard it is appropriate to consider the stack effects and resulting usefulness. I am also aware of a number of other Forth implementations that return the address of the count for this definition. Before making a formal proposal I would like to hear what others think about this.

```
7.0790   BLK
7.0800   BLOCK
7.1790   LOAD
```

Even though I find sequential file and stream I/O more useful for editing source and my applications, I read with interest the continued reference that "If `BLK` is zero, the input stream is being taken from `TIB`." And for `LOAD`, "...an exception exists if u is zero, or is not a valid block number."

Personally, this has never bothered me and seemed to offer consistency when thinking of virtual memory, even as a beginner, namely, that zero implied console input and any number greater than zero referred specifically to a *block* of virtual memory. And then along came F83 with its definitions of virtual memory, including the numbering of the first block as zero, which could be edited but not loaded. Whenever I asked people why this inconsistency, all I got were rationalizations about how it provided a great place for comments. Now really, a simple `-->` or `;S` at the beginning of any block allows the placement of comments, so why should I want to be restricted to one block of comments right at the beginning of the file: just to say "And the rest is silence"?

But I am still not clear from Basis 10 if `BLOCKS` will be numbered from zero up or from one.

To: Zafar Essak
From: R.Berkey [Robert]

ZE> "...feel strongly that if `"` returned the address of the count it would be more useful."

Yes, but address and length operands decouple the data structure from words that manage it, and allow one common, portable, general-purpose string descriptor. `SKIP` and `SCAN`, for example, must have address and length. My preference is to have one common string descriptor in the standard, and another name for the single operand `"`. Just last week I `EDITALL`ed through a megabyte of application code and renamed each of the `"` to `$"` in preparation for adding the address-and-length `"`.

ZE> "... I got...rationalizations how (block 0) provided a great place for comments."

The only Forth I've known that had blocks starting with one was my mistaken

(Continued on page 41.)

(Sort code continued from page 34.)

Baden's Basket Sort

```
: loc ( a -- a') cell+ ; ( When the next cell starts the data.)

( : loc    cell+ @ ;  ( When the next cell points to the data.)

256 constant M   ( # of "digits".)

: array ( k -- ) Create    cells allot
   ( i -- a) does> swap cells + ;

M array botm                       ( The bottoms of the sublists.)
M array top                        ( The tops of the sublists.)

( Knuth's Algorithm H reworked.)
: hook-up-queues ( link lim init -- tail)
   ( Hook the sublists back up.)
   DO ( link)
       I botm @ ( link link') ?dup
          IF    I top @ rot ! ( link') THEN
   LOOP ;

( The meat of Knuth's algorithm R.)
: sort-on-byte ( K h -- )
   ( h is head of a list.  K is byte in record to sort on.)
   0 botm M cells erase
 BEGIN ( K link)
     2dup loc + c@ ( K link i)
     dup botm @
     IF
         2dup botm @ !
     ELSE
         2dup top !
     THEN
        THEN
        over swap ( K link link i) botm ! ( K link)
        @
        dup 0=
   UNTIL   2drop ;

( The two foregoing definitions can be used as a foundation for
( basket sorting. Here they are combined to provide a general
( routine to sort on a field.)

( Algorithm R.)
: field-sort ( Head Field-start Field-end - )
    ( Given the head of a linked list and a field specification,
    ( for each byte in the field from least significant to most
    ( significant, separate the list into M sublists and then
    ( hook back up the sublists.  At the conclusion the list
    ( will be logically ordered by the field.)
    DO ( Head)
        I over @ ( Head    k P)
        sort-on-byte ( Head)
        dup M 0 hook-up-queues ( Head tail)
        0 swap ! ( Head)
    -1 +LOOP    drop ;

( ************************************************************ )

( This uses "sort-on-byte" and "hook-up-queues" for the data
( in the sort contest. "w@" and "w!" are for 2-byte data.)

Create Links   ITEMS CELL 2+ * allot    Variable Head

: Crown    Links Head !;

: Build-list
    Crown   Head @ ( P)
    ITEMS 0
```

(Sort code continues on page 41.)

FIG CHAPTERS REPORT

VISIT TO A PARALLEL UNIVERSE

JACK WOEHR - 'JAX' on GEnie

Forth is booming. The popular press has lost interest in Forth even to the extent of ritualistically announcing the Death of Forth. But in the ever-expanding field of embedded control programming, Forth has come into its own, joining the C language in what we, who know better from our own experience, might be otherwise tempted to call "the death of BASIC."

Yet where will the Forth programmers come from to maintain tomorrow the software being written today? Colleges don't exactly churn out trained Forthers in large numbers. The role of producing new Forth programmers is one that has traditionally been assumed by the Forth Interest Group and its members and affiliated chapters. The tremendous increase in accessibility of expert assistance in Forth via telecom now offers another path to progress for the would-be Forther. And every once in a while, a ray of bright light breaks through the perpetual gloom surrounding Forth's hind-teat position on the sow of Academia.

Joseph Gradecki is a senior at Metro Community College in Denver, Colorado. He has built a 16-node hypercube parallel processor out of Intel 8031 microprocessors. Each node has Forth in 8K EPROM and sports 32K static RAM. The system controller runs on an MS-DOS portable, which communicates serially with the nodes in a round-robin poll. The nodes communicate internally with each other in parallel along the edges of the hypercube via 8255 peripheral interface adapters, two per board.

Each node runs an identical Forth, which Mr. Gradecki wrote himself.

Mr. Gradecki and his Computer Science instructor, Dr. Charles P. Howerton, spoke at the January meeting of the Denver FIG Chapter, held at the National Institute of Standards and Technology before what (for Denver FIG) constitutes an overflow crowd, 26 souls all told. Joe's PPC (Personal Parallel Computer) is 16 wire-wrapped boards inside a tinted Plexiglass case. The cooling fan hums quietly and there are red LEDs flashing as each node wakes up. It's a veritable "black box." The PPC engages in distributed processing. Program and data are uploaded serially to the nodes from the system controller. Mr. Gradecki's chosen demo to the group features a keyed text search. Afterwards there are questions.

FD: How did you learn Forth?

Mr. Gradecki: "My two guidelines were fig-FORTH for the 8080 and a book called *Threaded Interpretive Languages* (Loeliger, Byte Books, 1981). I converted the floating point from 8086 code.

"Dr. Howerton got me interested in Forth. He gave me Brodie's book and got me started on the PC. To be truthful, this [demo] is the second Forth [application] program that I have ever written. I've spent so much time in the development that I haven't had time to play!

"My first Forth program was a Mandelbrot program. It pushed the limits of Forth. This code probably does too, but it was hacked together in the last few days. Time has been crucial."

FD: What has been your impression of Forth since you started to use it?

Mr. Gradecki: "I like it."

FD: What is the advantage? What is attractive about it?

Mr. Gradecki: "To me, it's simple. I can think in Forth very easily. Other people, I know, like things like C. I'm in the process of translating a ray-tracing program for the PPC from C. It's going to be real interesting. It's going to be real Forth; a case of 'let's see if this language can handle this.'

"With the cube here, when I was testing I didn't have to compile, link, handle the warnings... I just entered a definition and boom! It was just outstanding for that."

Dr. Howerton: "One of the reasons I suggested Forth to Joe was that he needed a bigger virtual environment in which to execute. He needed a richer instruction set without having to fall back to assembly language. So this way, by building a fundamental TIL, and then with the ability to outload definitions to it and expand it on the fly, he could build anything he wanted and it was simple."

FD: With the effort of implementing Forth on that chip, do you think that, in the end, you made a net memory savings in the program by using Forth rather than a straight assembly language program?

Dr. Howerton: "In terms of a new single program, probably."

Mr. Gradecki: "Definitely. I have the code on there pretty well optimized. In any event, it's a heck of a lot easier to program. Granted, I could write applications like this in assembler. I happen to enjoy assembler. It would have taken a lot longer, though. When I started, everything was in assembler. I didn't have the capability to upload to the processors. If I wanted something new in there, I burned it into EPROM."

Dr. Howerton: "From the time I gave him the books until he demonstrated Forth running on this thing, he took nine days. I thought he was faking it, but he brought up the interpreter."

FD: Is Forth something that you often bring up in your classes?

Dr. Howerton: "I do when I teach assembly language. I usually save the last three or four weeks of assembly language courses to teach Forth. The result is an immediate improvement in the students' assembly language skills. It's a relatively easy transition for the students, once they are at that point. Some terms, I teach VAX assembler; the term Joe took my course, I taught PC assembler. There are Forths for all of them, so it doesn't really matter. I can't think of a machine going that doesn't have a Forth for it. There's Forth for the 1802, Forth for the Cray..."

Mr. Gradecki's next project is ray tracing with his hypercube, an experiment in producing something "non-trivial" for his PPC to run. And he envisions building future machines, "...based on what I've learned implementing this, based on what I've seen elsewhere. I like hypercubes. The obvious thing is, more speed, more memory! Faster, better!"

To which the Forth programmer concerned with the maintainability of his or her code into the next century can only add this plea to the Dr. Howertons of America and their students, "More Forth programmers, please! Faster, better!"

(Continued from page 39.)

first attempt at a fig-FORTH. The problem only appears with operating-system Forths, as boot or object code gets put in block zero on standalone systems. At Dysan, we kludged around the problem by storing -1 in `BLK` when block zero was being interpreted. Pygmy's approach in mapping file blocks onto one master set of block numbers has a certain elegance.

ZE> "But I am still not clear from Basis 10 if `BLOCKS` will be numbered from 0 up or from 1."

Good point. X3/J14 has deleted the Forth-83 specifications that there are known block numbers 0–31, with other available block numbers documented. I'm hesitant to guess why.

To: Roedy Green
From: R.Berkey [Robert]

RG> "7.0340 `2>R` Remove the editor's note on why `DO` parms are backwards..."

I agree, even if for a different reason.

What the `2>R` rationale doesn't quite say is that `DO` has been implemented on pre-1983 systems as:

```
: DO
  COMPILE 2>R ;
  IMMEDIATE
```

with the parameters for `DO` having been set long before this 1981 implementation was noticed.

Upon learning of `2>R`, I at first thought that the parameters were backwards, a kludge to save a few bytes by not having a `(DO)`. Later, I realized that they aren't backwards. Looking at how the numbers appear in memory, this version maintains the double number. Another way to describe this is that `2>R` can be implemented using `2@` and `2!`.

If the rationale given in Basis was the reason for this order of parameters now being standardized, I'd be opposed to it: because standardizing a word designed for obsolete loops that speed-optimize on `DO` is standardizing garbage.

Anyway, thanks for your comments about the `2>R` rationale.

Figure One.

```
<word> ::= (<non-blank character> ... )
<Forth program> ::= { <Forth program> [<word> ...]}
```

Figure Two.

```
Program := { Word }
Word := ":" Identifier { Word | Number } ";"
:           :
:           :
```

(Sort code continued from page 39.)

```
    DO
        dup cell+ ( P addr-for-copy-of-data)
        I s@ over w!    2+ ( P next-P)
        dup rot ! ( P)
    LOOP
    2- CELL -    0 swap ! ;

: Reorder
    Head ( link) ITEMS 0
        DO   @   dup cell+ w@ I s!   LOOP   drop ;

( A trick to determine MSB of 2-byte cells.)
1 Pad w!   Pad c@ constant MSB#
         1 MSB# - constant LSB#

: Basket-sort
    Build-list
    Head 0 1 field-sort

    Reorder ;

: Elvey-sort
    Build-list
    LSB# Head @ ( k P) sort-on-byte (  )
    Head 256 0 hook-up-queues ( tail)
    0 swap ! (  )
    MSB# Head @ ( k P) sort-on-byte (  )
    Head ( link)
    256 128 hook-up-queues ( link)
    128 0 hook-up-queues ( tail)
    0 swap ! (  )
    Reorder ;
```

FIG CHAPTERS

The FIG Chapters listed below are currently registered as active with regular meetings. If your chapter listing is missing or incorrect, please contact Kent Safford at the FIG office's Chapter Desk. This listing will be updated in each issue of *Forth Dimensions*. If you would like to begin a FIG Chapter in your area, write for a "Chapter Kit and Application." Forth Interest Group, **P.O. Box 8231, San Jose, California 95155**

U.S.A.

- **ALABAMA**
  Huntsville Chapter
  Tom Konantz
  (205) 881-6483

- **ALASKA**
  Kodiak Area Chapter
  Ric Shepard
  Box 1344
  Kodiak, Alaska 99615

- **ARIZONA**
  Phoenix Chapter
  4th Thurs., 7:30 p.m.
  Arizona State Univ.
  Memorial Union, 2nd floor
  Dennis L. Wilson
  (602) 381-1146

- **ARKANSAS**
  Central Arkansas Chapter
  Little Rock
  2nd Sat., 2 p.m. &
  4th Wed., 7 p.m.
  Jungkind Photo, 12th & Main
  Gary Smith (501) 227-7817

- **CALIFORNIA**
  Los Angeles Chapter
  4th Sat., 10 a.m.
  Hawthorne Public Library
  12700 S. Grevillea Ave.
  Phillip Wasson
  (213) 649-1428

  North Bay Chapter
  2nd Sat., 10 a.m. Forth, AI
  12 Noon Tutorial, 1 p.m. Forth
  South Berkeley Public Library
  George Shaw (415) 276-5953

  Orange County Chapter
  4th Wed., 7 p.m.
  Fullerton Savings
  Huntington Beach
  Noshir Jesung (714) 842-3032

  Sacramento Chapter
  4th Wed., 7 p.m.
  1708-59th St., Room A
  Bob Nash
  (916) 487-2044

  San Diego Chapter
  Thursdays, 12 Noon
  Guy Kelly (619) 454-1307

  Silicon Valley Chapter
  4th Sat., 10 a.m.
  H-P Cupertino
  Bob Barr (408) 435-1616

  Stockton Chapter
  Doug Dillon (209) 931-2448

- **COLORADO**
  Denver Chapter
  1st Mon., 7 p.m.
  Clifford King (303) 693-3413

- **CONNECTICUT**
  Central Connecticut Chapter
  Charles Krajewski
  (203) 344-9996

- **FLORIDA**
  Orlando Chapter
  Every other Wed., 8 p.m.
  Herman B. Gibson
  (305) 855-4790

  Southeast Florida Chapter
  Coconut Grove Area
  John Forsberg (305) 252-0108

  Tampa Bay Chapter
  1st Wed., 7:30 p.m.
  Terry McNay (813) 725-1245

- **GEORGIA**
  Atlanta Chapter
  3rd Tues., 7 p.m.
  Emprise Corp., Marietta
  Don Schrader (404) 428-0811

- **ILLINOIS**
  Cache Forth Chapter
  Oak Park
  Clyde W. Phillips, Jr.
  (312) 386-3147

  Central Illinois Chapter
  Champaign
  Robert Illyes (217) 359-6039

- **INDIANA**
  Fort Wayne Chapter
  2nd Tues., 7 p.m.
  I/P Univ. Campus
  B71 Neff Hall
  Blair MacDermid
  (219) 749-2042

- **IOWA**
  Central Iowa FIG Chapter
  1st Tues., 7:30 p.m.
  Iowa State Univ.
  214 Comp. Sci.
  Rodrick Eldridge
  (515) 294-5659

  Fairfield FIG Chapter
  4th Day, 8:15 p.m.
  Gurdy Leete (515) 472-7077

- **MARYLAND**
  MDFIG
  Michael Nemeth
  (301) 262-8140

- **MASSACHUSETTS**
  Boston Chapter
  3rd Wed., 7 p.m.
  Honeywell
  300 Concord, Billerica
  Gary Chanson (617) 527-7206

- **MICHIGAN**
  Detroit/Ann Arbor Area
  Bill Walters
  (313) 731-9660
  (313) 861-6465 (eves.)

- **MINNESOTA**
  MNFIG Chapter
  Minneapolis
  Fred Olson
  (612) 588-9532

- **MISSOURI**
  Kansas City Chapter
  4th Tues., 7 p.m.
  Midwest Research Institute
  MAG Conference Center
  Linus Orth (913) 236-9189

  St. Louis Chapter
  1st Tues., 7 p.m.
  Thornhill Branch Library
  Robert Washam
  91 Weis Drive
  Ellisville, MO 63011

- **NEW JERSEY**
  New Jersey Chapter
  Rutgers Univ., Piscataway
  Nicholas Lordi
  (201) 338-9363

- **NEW MEXICO**
  Albuquerque Chapter
  1st Thurs., 7:30 p.m.
  Physics & Astronomy Bldg.
  Univ. of New Mexico
  Jon Bryan (505) 298-3292

- **NEW YORK**
  Rochester Chapter
  Odd month, 4th Sat., 1 p.m.
  Monroe Comm. College
  Bldg. 7, Rm. 102
  Frank Lanzafame
  (716) 482-3398

- **OHIO**
  Cleveland Chapter
  4th Tues., 7 p.m.
  Chagrin Falls Library
  Gary Bergstrom
  (216) 247-2492

- **Columbus FIG Chapter**
  4th Tues.
  Kal-Kan Foods, Inc.
  5115 Fisher Road
  Terry Webb
  (614) 878-7241

  Dayton Chapter
  2nd Tues. & 4th Wed., 6:30 p.m.
  CFC. 11 W. Monument Ave. #612
  Gary Ganger (513) 849-1483

- **OREGON**
  Willamette Valley Chapter
  4th Tues., 7 p.m.
  Linn-Benton Comm. College
  Pann McCuaig (503) 752-5113

- **PENNSYLVANIA**
  Villanova Univ. Chapter
  1st Mon., 7:30 p.m.
  Villanova University
  Dennis Clark
  (215) 860-0700

- **TENNESSEE**
  East Tennessee Chapter
  Oak Ridge
  3rd Wed., 7 p.m.
  Sci. Appl. Int'l. Corp., 8th Fl.
  800 Oak Ridge Turnpike
  Richard Secrist
  (615) 483-7242

- **TEXAS**
  Austin Chapter
  Matt Lawrence
  PO Box 180409
  Austin, TX 78718

  Dallas Chapter
  4th Thurs., 7:30 p.m.
  Texas Instruments
  13500 N. Central Expwy.
  Semiconductor Cafeteria
  Conference Room A
  Clif Penn (214) 995-2361

  Houston Chapter
  3rd Mon., 7:30 p.m.
  Houston Area League of PC Users
  1200 Post Oak Rd.
  (Galleria area)
  Russell Harris
  (713) 461-1618

- **VERMONT**
  Vermont Chapter
  Vergennes
  3rd Mon., 7:30 p.m.
  Vergennes Union High School
  RM 210, Monkton Rd.
  Hal Clark (802) 453-4442

- **VIRGINIA**
  First Forth of Hampton Roads
  William Edmonds
  (804) 898-4099

  Potomac FIG
  D.C. & Northern Virginia
  1st Tues.
  Lee Recreation Center
  5722 Lee Hwy., Arlington
  Joseph Brown
  (703) 471-4409
  E. Coast Forth Board
  (703) 442-8695

  Richmond Forth Group
  2nd Wed., 7 p.m.
  154 Business School
  Univ. of Richmond
  Donald A. Full
  (804) 739-3623

- **WISCONSIN**
  Lake Superior Chapter
  2nd Fri., 7:30 p.m.
  1219 N. 21st St., Superior
  Allen Anway (715) 394-4061

INTERNATIONAL

- **AUSTRALIA**
  Melbourne Chapter
  1st Fri., 8 p.m.
  Lance Collins
  65 Martin Road
  Glen Iris, Victoria 3146
  03/29-2600
  BBS: 61 3 299 1787

  Sydney Chapter
  2nd Fri., 7 p.m.
  John Goodsell Bldg., RM LG19
  Univ. of New South Wales
  Peter Tregeagle
  10 Binda Rd.
  Yowie Bay 2228
  02/524-7490
  Usenet
  tedr@usage.csd.unsw.oz

- **BELGIUM**
  Belgium Chapter
  4th Wed., 8 p.m.
  Luk Van Loock
  Lariksdreff 20
  2120 Schoten
  03/658-6343

  Southern Belgium Chapter
  Jean-Marc Bertinchamps
  Rue N. Monnom, 2
  B-6290 Nalinnes
  071/213858

- **CANADA**
  BC FIG
  1st Thurs., 7:30 p.m.
  BCIT, 3700 Willingdon Ave.
  BBY, Rm. 1A-324
  Jack W. Brown
  (604) 596-9764
  BBS (604) 434-5886

  Northern Alberta Chapter
  4th Sat., 10a.m.-noon
  N. Alta. Inst. of Tech.
  Tony Van Muyden
  (403) 486-6666 (days)
  (403) 962-2203 (eves.)

  Southern Ontario Chapter
  Quarterly, 1st Sat., Mar., Jun., Sep., Dec., 2 p.m.
  Genl. Sci. Bldg., RM 212
  McMaster University
  Dr. N. Solntseff
  (416) 525-9140 x3443

- **ENGLAND**
  Forth Interest Group-UK
  London
  1st Thurs., 7 p.m.
  Polytechnic of South Bank
  RM 408
  Borough Rd.
  D.J. Neale
  58 Woodland Way
  Morden, Surry SM4 4DS

- **FINLAND**
  FinFIG
  Janne Kotiranta
  Arkkitehdinkatu 38 c 39
  33720 Tampere
  +358-31-184246

- **HOLLAND**
  Holland Chapter
  Vic Van de Zande
  Finmark 7
  3831 JE Leusden

- **ITALY**
  FIG Italia
  Marco Tausel
  Via Gerolamo Forni 48
  20161 Milano
  02/435249

- **JAPAN**
  Japan Chapter
  Toshi Inoue
  Dept. of Mineral Dev. Eng.
  University of Tokyo
  7-3-1 Hongo, Bunkyo 113
  812-2111 x7073

- **NORWAY**
  Bergen Chapter
  Kjell Birger Faeraas,
  47-518-7784

- **REPUBLIC OF CHINA**
  R.O.C. Chapter
  Chin-Fu Liu
  5F, #10, Alley 5, Lane 107
  Fu-Hsin S. Rd. Sec. 1
  TaiPei, Taiwan 10639

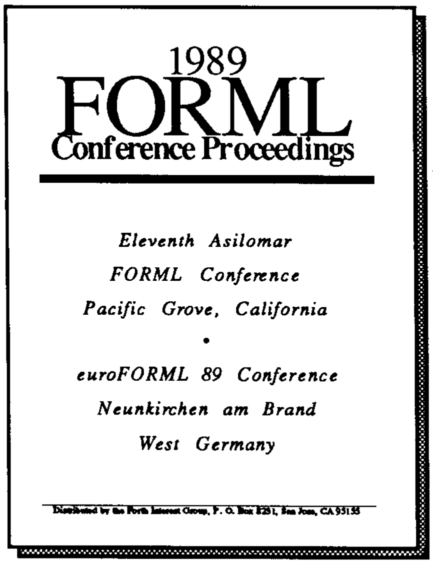
- **SWEDEN**
  SweFIG
  Per Alm
  46/8-929631

- **SWITZERLAND**
  Swiss Chapter
  Max Hugelshofer
  Industrieberatung
  Ziberstrasse 6
  8152 Opfikon
  01 810 9289

- **WEST GERMANY**
  German FIG Chapter
  Heinz Schnitter
  Forth-Gesellschaft C.V.
  Postfach 1110
  D-8044 Unterschleissheim
  (49) (89) 317 3784
  Munich Forth Box:
  (49) (89) 725 9625 (telcom)

SPECIAL GROUPS

- **NC4000 Users Group**
  John Carpenter
  1698 Villa St.
  Mountain View, CA 94041
  (415) 960-1256 (eves.)


1989
FORML
Conference Proceedings

Eleventh Asilomar
FORML Conference
Pacific Grove, California
•
euroFORML 89 Conference
Neunkirchen am Brand
West Germany

Distributed by the Forth Interest Group, P. O. Box 8231, San Jose, CA 95155

Forth Modification Laboratory

FORML and euroFORML

Conference Proceedings

Conference papers from the eleventh annual FORML conference held November 24-26, 1989 at the Asilomar Conference Center, Pacific Grove, California, U.S.A. and conference papers from the euroFORML 89 conference held October 13-15, 1989 at Neunkirchen am Brand, Federal Republic of Germany. 446 pages.

$40.00

TABLE OF CONTENTS

FORML

Forth Poetry or Forth Program?
A Trail of Bread Crumbs
Seeing Forth
From Pascal to Forth
Binary Radix Sort on 80286, 68010, and RTX2000
An Extensible Optimizer for Compiling Forth
Four Different Programmers, Forths, and Computers
Multitasking or Multiple State Machines
Graphics Based Smart Windows
A State Machine Based Drawing Package
Communications and State Machines
A 3-D Measurement System Using Object-Oriented Forth
User-Defined Systems for Pure Mathematics
CRC Polynomials Made Plain
Hierarchical Objects from Flat Vocabularies
PAI Virtuoso
Programmable Controlled Processing And Graphic System
Pattern Matching in Forth
Control Flow Words From Basis9
(FPC) Forth for the PC
Logic Stack
A Cross-Assembler for a Small Interactive Target
A Stack Machine Assembler
An Object-Oriented Forth Implementation
For think Forthink Forth Ink
Smart Ram
Thermal Meter—An Application of FORTH FSY63
The Harris C Cross Compiler
Adding Compiler Security to METHODS>
Proposal for Syntax Notation
Cool-Unifying Class and Prototype Inheritance

euroFORML '89

A Modular Approach to Robotic Control Systems
Two Levels of Parallelism or a New Approach for Control Systems Design
Software Development System USW
Zug Power Station, An Application of Trainable Neural Nets
Memory Cards and Forth: An Update
A Comparison of the Cooperative and Pre-emptive Concurrent Scheduling Algorithms
XShell: A Cross-development User Interface
dbgx - The Harris RTX 2000 C Language Debugger
Module Forth
An RTX Recompiler for On-Line Maintenance
A Vision of Transparency
A Databased Forth
Interactive Remote Compilation for Development and Machine Integration
Forth and Ocean Bottom Seismograph
Have Dot-If Dot-Else Dot-Then
Programming and Our View of Man
Some Experiences on Implementing Floating-Point in Forth
RTX 4000
Object Oriented Menu Creating System for Interactive Forth Applications
From Block Files to the Twentieth Century
Not Screens Nor Files But Words
The search for the Law

FIG members entitled to membership discount–see order form inside.

Forth Interest Group
P.O. Box 8231
San Jose, CA 95155

Second Class
Postage Paid at
San Jose, CA