VOLUME XII, NUMBER 5 JANUARY/FEBRUARY 1991 $6.00

FORTH DIMENSIONS

INTERRUPT HANDLING IN FORTH

INTERRUPT-DRIVEN COMMUNICATIONS

EAR TRAINING

GENIE FOR BEGINNERS

68000 NATIVE-CODE FORTH (IV)

SILICON COMPOSERS INC

Introducing the

SC/FOX CUB^{tm} 2000/2001A

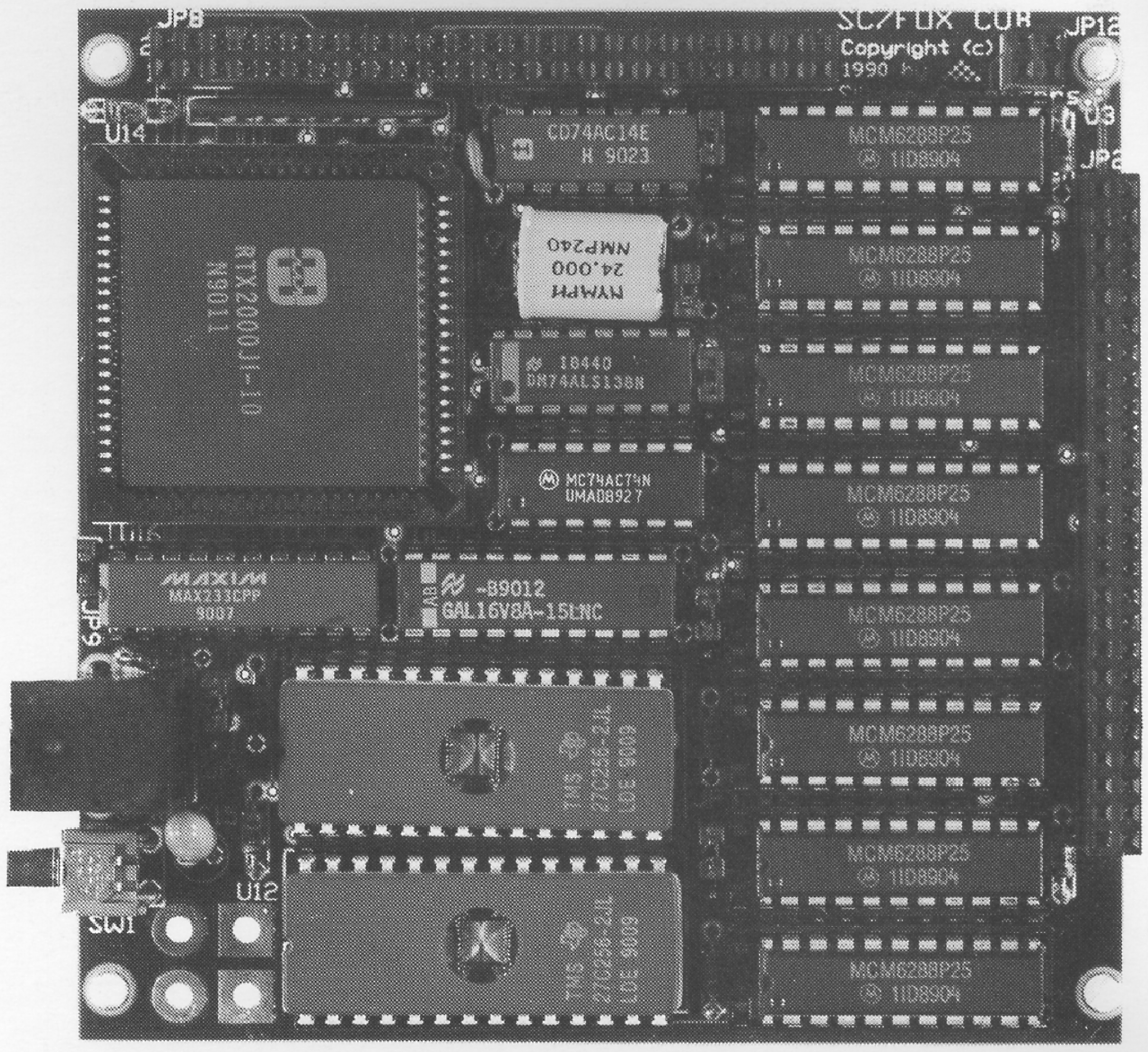
Single Board Computer

Board Features:
- Uses RTX 2000^{tm} or 2001A PLCC chip.
- 16, 20, or 24 MHz input clock operation.
- 100mm by 100mm Eurocard-size board.
- High quality 4-layer board with full power and ground planes for reduced noise.
- 32K to 256K bytes 0-wait-state SRAM.
- 64K bytes shadow EPROM.
- RS232 serial port, up to 1.25M baud.
- +5 volt operation: 145 mA @ 8 MHz; 200 mA @ 10 MHz; 230 mA @ 12 MHz.
- Fully assembled and tested, ready to use.

Development Software Included:
- FCompiler Language, SC/Editor, SRUN I/O Server Utility (CUB 2000).
- SC/Loader in Shadow EPROM (CUB 2000).
- cmForth (full source code) & SCX Server/Block Editor (CUB 2001A and CUB 2000).
- Comprehensive manual and program examples.

Optional SC/FOX CUB Products:
- SC/Forth^{tm} Language in EPROM.
- SC/Float^{tm} Floating Point Library.
- SC/SBC/PROTO^{tm} Prototype Board.

Harris RTX 2000/2001A Forth Processors:
- RTX 2000 or RTX 2001A in CMOS PLCC package.
- 1-cycle 16x16, 32-bit on-chip multiplier (RTX 2000 only).
- Two 256-word (16-bit) stack memories (RTX 2000 only).
- Two 64-word (16-bit) stack memories (RTX 2001A only).
- 14-prioritized interrupts, 4 cycle latency.
- Three on-chip 16-bit timer/counters.
- 8-channel multiplexed 16-bit I/O bus.
- 83ns clock cycle at 12 MHz.



SC/FOX CUB^{tm} Single Board Computer
(actual size)

The SC/FOX CUB Single Board Computer is a small high-speed, stand-alone computer board, ideal as a low-power/cost choice for evaluation or for applications such as embedded-systems control, data acquisition, instrument control, image and numerical processing, process control, factory automation, machine control, inspection systems, robotics, and real-time systems control.

ONE-TIME INTRODUCTORY OFFER
(Good Through 3/1/90)
$450 plus tax and shipping with Prepaid Orders

For additional product information and OEM pricing, please contact us at:

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

FORTH DIMENSIONS

EDITORIAL

Dissent ≠ Exile

In these pages, dissent does not mean exile. Readers who have been following the development of an ANS Forth will be most interested in our "Letters" department. It contains a summary of the objections and philosophical differences held by a vocal and diligent group of dissidents at the Boston FIG Chapter. While their views may not mirror those of every anti-ANSI activist, and while their letter may not delineate all the details of their own arguments, it does give voice to those who prefer a minimalist Forth and who fear that ANS Forth will be more extravagant than economical, that it will break new ground instead of mapping well-tested turf.

Luckily for us, members of ANS X3J14 (the committee that is developing ANS Forth, which met recently in Detroit) got wind of the fact that Boston FIG's letter would be appearing here. By stretching our deadlines just a bit, we were able to print their response in the same issue, along with the formal statement of their scope of work.

All the above is printed here in its entirety, because we believe that all manner of voices in the Forth community should be heard and understood if we are to achieve any kind of collective wisdom. I am sure that ANS Forth will not manage to be all things to all people—some of whom decided in advance that it would mean nothing at all to them, regardless of its content. But it will be the standard most looked upon by the outside world and, to some degree, every Forth implementation will be understood relative to it. Except for projects that require strict adherence to an ANS standard programming language, whether a vendor adheres to ANS Forth will not matter as much as understanding what it represents as a whole and one's own reasons for adopting or rejecting its specifics.

*   *   *

But plenty of other material is here to be mulled over. Interrupt handling is approached from both application-specific and more general directions. We hope these articles spur input on the subject from other readers, in the tradition of working together to build upon, diverge from, preempt, and refine published works. In fact, author Streed proposes a collective effort to develop a communications application in Forth. He points out that we could then more easily add all the bells and whistles we wish the makers of our "com" programs had provided.

Speaking of which, if you haven't yet gotten on-line with the Forth Interest Group on GEnie, or if you tried but found the system too opaque and mainframely for your tastes, check "GEnie for Beginners." It should help smooth the way. Even speaking as an occasional user, and one long accustomed to graphical interfaces at that, my on-line time has been well worth while: both interesting and labor-saving in the long run.

Finally, many of you have been following John Redmond's ForST series with interest. Originally, we debated publishing articles about a new Forth for the Atari ST; but we found the author's ideas fascinating and potentially of general relevance. What was to have been three articles has become five (concluding in the next issue). In this fourth installment, we are treated to Redmond's implementation of register variables and are shown quite explicitly the benefits that accrue from their use.

—Marlin Ouverson
Editor

P.S. If you enjoy quick glimpses of Forth in high places—or very, very deep ones—be sure to read the biographical statement following Barrie Walden's article, "Forth Interrupt Handling."

Forth Dimensions
Published by the
Forth Interest Group
Volume XII, Number 5
January/February 1991
Editor
Marlin Ouverson
Advertising Manager
Kent Safford
Design and Production
Berglund Graphics
Circulation/Order Desk
Anna Brereton

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."

LETTERS

More Reliable 80x86 Division
Dear Editor:

David Arnold's article, "Reliable 8086 Division," in the November/December issue of *Forth Dimensions*, contains an interesting and useful discussion of the Forth division operators. However, David's information about Interrupt 0 handling on Intel 80x86-family microprocessors appears to be incomplete. The 80286's Interrupt 0 behavior, which puts the address of the divide instruction which caused the interrupt on the stack, is also the behavior of all subsequent Intel microprocessors (80386X, 80386DX, and 80486). This change from the behavior of the 8086/88 is well documented by Intel and (as the time-honored phrase goes) is not a bug but a feature, in that it allows an Interrupt 0 handler to "fix up" the arguments which caused the exception and then restart the failing instruction.

Regards,
Ray Duncan
Laboratory Microsystems, Inc.
P.O. Box 10430
Marina del Ray, California 90295

Dear Marlin,

As I was checking my article, "Reliable 8086 Division," I found a couple of errors that were in the manuscript provided to you. I beg pardon for letting them get past me.

1. On page eight, in paragraph four, which begins some discussion of floored and unfloored division, I incorrectly wrote that school children are taught floored division. I should have said *unfloored* division.

2. On page twelve, in screen 12, line 2, the source code listing incorrectly says this:

```
AX AX XOR  DIV_ERR? #) AX MOV
```

It should have said this:

```
AX AX XOR  AX DIV_ERR? #) MOV
```

This bit of code is supposed to load a default false value into the division error flag at the beginning of a `/MOD` division process. If an error later occurs during hardware division, a special interrupt handler updates the flag with a true value. The incorrect code doesn't initialize the flag. The would not result in incorrect division, because a specious result in this version of `/MOD` would be re-done with an error-free auxiliary routine. It would waste a little time, though. Here is what can happen:

The incorrect version leaves unchanged the initial contents of the error-flag register, `DIV_ERR?`. If a preceding division resulted in a division error, a true value would remain. Hardware division then takes place, and the flag value is checked to see if an error occurred. If true, the auxiliary process of `SAFE_*/MOD` re-does the arithmetic and leaves a valid flag.

As may be seen, the flag would finally get reset, but in an awfully roundabout way.

Yours truly,
David Arnold
616 1/2 W. Hamilton St.
Kirksville, Missouri 63501

Catch & Throw
Doug Phillips:

I saw your note in "Best of GEnie" (*FD* XII/3). I had no idea `CATCH/THROW` had caught on since I recommended the mechanism to George Shaw last year. Your "for free" analysis of the requirements for `CATCH/THROW` implementation is not quite complete. There is a middle ground between your two approaches. I have used this technique for years in my Forth systems, and the mechanism is free if your program doesn't use it, and quite cheap if it does.

The idea is to have a pointer to the topmost catch frame on the return stack, and links from catch frame to catch frame. So far, this is just like your first choice. Now, when `CATCH` is executed, it builds the catch frame and *calls* the remainder of its containing routine (instead of returning). When the containing routine returns to the `CATCH` via `NEXT`, `CATCH` simply removes the catch frame and falls into `NEXT`, too, returning from the containing routine. `NEXT` itself is not changed in the least! The return stack can have anything at all on it (in fact, I use it for my `LOCAL` mechanism).

Now `THROW` is implemented as a return from the outermost `CATCH` (which hasn't returned yet... it called its return point) after restoring the catch frame pointer. I also have a routine called `PUNT-CATCH` which removes the topmost catch frame from the return stack.

Reproduced in Figure One are the `CATCH`, `THROW`, and `PUNT-CATCH` routines in M68000 assembly code (this is native-code Forth, i.e., `NEXT` = RTS). The references to the frame pointer (`A2`) are for local variables.

As a further optimization, I have two mechanisms to set up local variables. One allows `CATCH`es in the routine with locals; the other doesn't, and is very cheap. The two mechanisms may be mixed in one program. This is just one more way to make `CATCH/THROW` inexpensive when not used.

Regards,
Doug Currie
Flavors Technology, Inc.
3 Northern Blvd.
Amherst, New Hampshire 03031

Open Letter to ANSI X3J14

[The following is a letter that was addressed to Elizabeth Rather, in her capacity as chairperson of the group that is formulating an ANS Forth. Its author also wished to share it with the readers of FD. *Following it is Ms. Rather's response.]*

This is an open letter to the members of ANSI ASC X3 / X3J14 addressed to you, the chair. I would like to thank you and X3J14 for the opportunity you afforded the Boston FIG ANS Forth Group, and me as their representative, to air our views and act upon our proposals at your recent (thirteenth) meeting in British Columbia.

Our group had hoped to sway X3J14 to our point of view—a so-called "minimalist" point of view—but the only proposals of ours that passed were either not controversial at all (post), or fit the already existing views of the current members of X3J14. The *Thirteenth Meeting of X3J14* was therefore a disappointment to us.

To be fair, there were a few small victories that must be mentioned with the casualties. X3J14's treatment of division and NOT represent compromise between Forth-79 and Forth-83. X3J14's passage of my motion to the technical committee (TC) makes it clear in the *Scope of Work for X3J14* that the lack of this or that whizzy feature is not to be considered a "problem area" (though an amendment stating "...unless deemed indispensable to the production of a coherent standard" significantly weakened the wording). And BASIS *did* get smaller, if only by one word.

My mission, however, was to try to change the "world view" of the current members of X3J14 and in that, I failed. This letter is an attempt to better explain our point of view and to sway the current membership of X3J14 to it.

At our most recent meeting on September 5th the discussion focused on the question, "why don't they understand our point of view and act on it?" To that end, the group came up with a way of understanding the standards process that we hadn't thought of before: "the three Cs"—Completeness, Compatibility, and (self-) Consistency. Completeness refers to ANS Forth specifying a language complete enough to be useful without adding extra features. Compatibility refers to ANS Forth being compatible with accepted practice. Consistency refers to the wording of the ANS Forth BASIS document being self-consistent.

It first appears obvious that "the three Cs" are each goals that ANS Forth should approach as closely as possible, but a second look reveals that significantly attaining some goals necessitates compromise on the others. We feel it best to compromise completeness, while the current members of X3J14 continually compromise compatibility with existing practice and apparently want ANS Forth to be a specification for the ultimate, complete Forth. It is our belief that *the vendors* are responsible for providing complete Forths and that the standards process should provide the Forth community at large with a standard document (not a specification) that describes the Forth that is compatible with accepted practice.

Forth is, after all, one of the few extensible languages. It is not necessary to put every language extension into standard Forth. It is only necessary that standard Forth provide the facilities for extending itself, so that users (and vendors) can add any language extension they want.

We believe that trying to specify every nook and cranny of a complete Forth system—especially in new areas that are outside accepted practice—is a process that is doomed to failure. Any specification written describing what Forth ought to be, rather than what Forth is, is bound to have holes in it. It is the usual fate of most well-meaning specification writers and it was the fate of the process that yielded Forth-83. X3J14 must standardize last year's Forth, not next year's Forth.

It is the belief of the Boston FIG ANS Forth Group that our point of view, while not well represented among the current members of X3J14, is prevalent in the Forth community at large. We will continue to drum up support for our point of view outside of X3J14 and continue to attempt to win over the current membership of X3J14 to that view by submitting proposals and comments.

I close with a quotation from Chuck Moore that is appropriate to the compelling sense of rightness our group recognizes in the minimalist point of view:

One principle that guided the evolution of Forth and continues to guide its application is, bluntly: Keep it simple. A simple solution has elegance. It is the result of exacting effort to understand the real problem and is recognized by its compelling sense of rightness. I stress this point, because it contradicts the conventional view that power increases with complexity. Simplicity provides confidence, reliability, compactness, and speed.

Sincerely,
David C. Petty
Boston Forth Interest Group
American National Standard Forth Group
P.O. Box 2
Cambridge, Massachusetts 02140-0001

X3J14 Report
Elizabeth D. Rather, Chair, X3J14

The primary focus of this article is to respond to concerns expressed by Boston FIG regarding our work towards ANS Forth. First, though, I'd like to give a little background.

X3J14 has held 14 meetings, the most recent of which was in Detroit Nov. 7–11. A total of 36 people have participated as Principal voting members (of which 21 are currently voting members), and an equivalent number have also participated actively as alternates or observers. Of these, 15 list themselves as producers (including FORTH, Inc., CSI, LMI, Harris, Vesta, Johns Hopkins, and a number of individual producers), 20 as consumers, and one as a general interest group (FIG). Several of our members are well known in the Forth community, including George Shaw, Mitch Bradley, Bob Berkey, Bill Ragsdale, Larry Forsley, and Martin Tracy. Some represent large organizations: NASA, IBM, NCR, Ford Motor Co.; while others are individual consultants. Some of our members are extremely highly skilled systems programmers, with extensive knowledge of Forth internals, while some are relatively casual users who are keenly interested in this process. Some are experienced with only one vendor's system, while others have used several. Similarly, some are expert programmers in a number of languages and OSs, while others aren't. In short, it's a very diverse group. In addition to our members, nearly 100 people have purchased at least one copy of our working BASIS documents; of these, most have bought several, and about 30 are subscribers.

We have met in many places: Melbourne, Florida; Washington, D.C.; Rochester; Boston; Detroit; Vancouver; Portland; San Jose; Palo Alto; Los Angeles; and San Diego. In 1991 we may add Atlanta and Boulder, Colorado. In all of these places we've invited area Forth users to attend, contribute, and vote in TSC sessions (see below), and many have done so.

Since our formation in August, 1987, we've had ten four-day meetings and four five-day meetings. With an average attendance of about 12 members and five visitors, that represents about 1,020 work days, many of which were 10–12 hours long, or roughly 4.25 years of work. In addition, we have poured many hours of work into proposals and study between meetings.

We've processed 958 proposals: 518 passed, 301 failed, 110 were withdrawn (because they were redundant or dealt with issues that had already been decided), 11 were declared comments, and 18 are still pending.

This is a lot of work, by a lot of very bright, dedicated people.

Here is how we work. The TC is governed by strict rules laid down by our governing organization, X3 (Information Processing Standards group of ANSI), which require, among other things, that every technical decision represent a consensus of the members. In order to arrive at that consensus, we've set up a sub-group, called the Technical Subcommittee, or TSC. Greg Bailey is its chair. It consists of whoever is present at a meeting: members, alternates, visitors, each with one vote. This group debates issues, often at great length, until it reaches a consensus, and then forwards that decision to the TC for official action. Usually the consensus survives in the TC, with most votes being overwhelmingly in support of the TSC's decision. Sometimes, however, new facts or questions arise about a proposal or issue, in which case we refer it back to the TSC. No proposal is permitted to pass or fail with a significant minority opinion.

In practice, this has ensured that there are no hasty or casual decisions, and no one's "private agenda" can prevail in the absence of overwhelming support.

At our first meeting, we adopted a statement of our Scope of Work, a step required by X3. A copy is attached. This was reexamined at length in May and August of this year [1990], and amended somewhat for clarity. In summary, this charter called for us to examine "common existing practices" in Forth, as well as identifying significant "problem areas" and attempting to resolve them. Neither of these has been easy.

What is "common existing practice?" Forth-83? Forth-79? One implementation that happens to have thousands of users? One brutal truth is that if we adopted a guideline that required all major implementations to agree, we'd have a subset of Forth that would run only on a 16-bit engine and have a command set so limited that no significant programs could be written with it. We have consistently and unanimously rejected this as a guideline.

One of our earliest acts was to identify vendors with over 200 users, and send them a questionnaire soliciting information as to which, if any, standard they followed and what they considered to be major problem areas that needed to be addressed. We also solicited input from many other sources, including FIG chapters, electronic bulletin boards, customers of member vendors, etc. We found very broad compliance with Forth-83, a significant minority of Forth-79 compliance, and some specific areas of concern, such as need to run on other than 16-bit architectures (especially 32-bit systems), need to deal with host OSs, floating-point arithmetic, etc.

There were many others. The implementors we studied had virtually all tackled these issues, with predictably diverse approaches. In order to resolve this diversity, we adopted a guideline:

If we changed the behavior of a word from its meaning in Forth-83, Forth-79, or significant current usage, we'd give it a new name. This prevents "breaking" existing code, as users wishing to comply with ANS Forth can freely choose to either:

a) Do a blanket name change, if they comply with the meaning;
b) Keep the old name and meaning, not changing existing programs, and add the new name and meaning for later use;
c) Add a "shell" on top of a system, defining the new word in terms of the existing word; or
d) Add a shell under an application, defining the old word in terms of the new word.

X3, which has been through these wars before (thirty years of Fortran and COBOL standards, for example), is very concerned about "cost of compliance," and as most of us operate on tight budgets, we heartily agree. All of these approaches are far cheaper than examining all instances of a word and deciding whether the usage is impacted by the change of meaning. However, this has been a source of some "new" word names which have been adopted to resolve important usage conflicts (e.g., `NOT`) and technical problems (e.g., usage of `COMPILE` and `[COMPILE]`).

We had many requests for things that were clearly new as far as Forth standards were concerned, but with which several implementors and users had extensive experience. In some of these cases we synthesized this experience, and then commissioned our members to try the synthesis and report results. This has been the case with all the optional word sets.

Our status at this time is that we're about ready to publish a draft proposed standard or "dpANS" for review, probably shortly after our next meeting (January 29–February 3, 1991, at FORTH, Inc.). The rules governing this phase of our activity are as follows:

1. The TC must approve the dpANS by a two-thirds vote of all members (taken by mail). Negative votes must be responded to in writing and will be kept with the document through all the following steps.
2. The dpANS is then submitted to X3's Standards Planning and Requirements Committee (SPARC) for review, which will take several months.
3. The dpANS is then published for public review, for a four-month period. Public review comments are sent to X3, and are tracked by them. The TC must respond in writing to all adverse public-review comments, and the responses are reviewed by X3 to ensure that we're truly responsive.
4. If, as a result of input from X3 or the public, we elect to make changes, the revised dpANS then goes out for additional two-month public review periods

```
    EXPORT  catch
 catch  ; ( - n T or F )  catch
    MOVE.W   #0, -(A4)           ; rtn val
    MOVEA.L (SP), A0                ; rtn addr, leave it for throw
    MOVE.L   A2, -(SP)            ; save frame ptr
    MOVE.L   _catches_(A6), -(SP)    ; setup catch frame
    MOVE.L   SP, _catches_(A6)          ; link it in
    JSR    (A0)                  ; return first time
    ; colon word RTS comes here
    MOVE.L   (SP)+, _catches_(A6)     ; unlink
    ADDQ.L   #8, SP                 ; remove catch frame
    RTS                         ; return from colon word

    EXPORT throw
 throw  ; ( n - n T )  throw - doesn't return to caller but to catch
    MOVE.W   #-1, -(A4)          ; rtn val T
    MOVE.L   _catches_(A6), SP      ; get catch frame
    MOVE.L   (SP)+, _catches_(A6)      ; unlink it
    MOVE.L   (SP)+, A2              ; restore frame ptr
    RTS                          ; return from catch again

    EXPORT punt_catch
 punt_catch ; ( - ) removes catch frame
    MOVEA.L (SP)+, A0               ; rtn addr

    ADDQ.L   #4, SP            ; remove catch's rtn addr
    MOVE.L   (SP)+, _catches_(A6)     ; unlink
    ADDQ.L   #8, SP              ; remove catch frame
    JMP     (A0)                   ; return
```

as often as needed until all issues are resolved to the satisfaction of both the TC and X3.

5. Finally, the dpANS plus all unresolved adverse comments and/or negative votes by TC members goes to X3 for review and final approval.

Most of our remaining work involves adding rationales and explanatory materials to help people understand not only what the Standard says but why we did what we did. We do have what we consider to be good reasons for each addition, subtraction, and change; and we are attempting to articulate these reasons as clearly as we can. There's a little remaining technical work on multitasking, number conversions, and search order.

Now, with this background, I'd like to comment on the specific concerns of the Boston FIG group.

These people have been among our most active and dedicated outside contributors. They've met regularly, and reviewed our work carefully and diligently. They've sent us extensive notes and comments, all of which have been distributed within the TC and read carefully by most of us. They've also submitted 18 proposals, of which ten have passed (some amended) and eight failed.

Their underlying concern is with the overall size of the standard, especially the required CORE word set. We generally agree with many of their viewpoints, perhaps more than they realize. But the devil is in the details: just what is the minimum useful word set? Our definition of it is represented by CORE, 135 words (compared with 132 in Forth-83). BFIG has proposed dropping such words as `1+`, `1-`, `2@`, `2!`, `2DROP`, `2DUP`, `2OVER`, `2SWAP`, `MAX`, `MIN`, `SPACE`, and others that have been in every standard and virtually all implementations. We simply feel this is carrying minimalism too far.

No one on the TC believes we are even close to defining "a complete Forth," and we agree that this is principally an implementor's task. We have rejected many proposed additional words and word sets, submitted both by members and outside observers. In fact, over two-thirds of our outside proposals have offered additional words, many of which have merit but were rejected as being outside common practice.

We are grateful for all the outside help we've received, from BFIG and others, and hope it will continue through the review process. We urge as many people as possible to participate, by buying copies of our current BASIS (send checks for $10 made to the Forth Vendor's Group, c/o FORTH, Inc., 111 N. Sepulveda, Manhattan Beach, CA 90266), downloading BASIS (now published in massive RTF files on GEnie and other boards), attending our next meeting (contact me at 1-800-55-FORTH for more information), and sending us your proposals and comments.

Resolution 87-002
Revision #2
Scope of Work for X3/J14

The purpose of this resolution is to outline the scope of work for this TC. It is based upon the project proposal adopted by X3J14/005. The intent is to present an outline of the significant steps to be followed to achieve an acceptable standard which will result in broad compliance among all major vendors of Forth language products, with minimum adverse impact upon transportability from existing systems in use.

The scope of work for X3/J14 shall encompass the following:

1. Identification and evaluation of common existing practices in the area of the Forth programming language. This shall include the following:
a. Identification of all producers of Forth language programming systems with a distribution in excess of 200 users.
b. Evaluation of Forth implementations distributed by these producers with respect to the Forth-83 standard, to identify the primary areas of non-compliance. Areas in which most producers are in compliance, or in agreement on a concept outside of the scope of the Forth-83 Standard, will be considered to be "accepted practice."
c. Public solicitation from these producers as well as other sources represented on the TC of specific problem areas within the Forth-83 Standard, and recommendations for change. Problem areas are areas of accepted practice where producers' implementations vary. Problem areas specifically do not include concepts new to Forth intended to improve perceived deficiencies in Forth as defined by accepted practice, unless deemed indispensable to the production of a coherent standard.
2. Evaluate proposed modifications to the Forth-83 Standard resulting from Item 1c above, addressing the following areas:
a. Arithmetic and logical operators
b. Flow-of-control structures
c. Input and output operators
d. Memory and mass storage operators

(Continued on page 17.)

SIGFORTH '91

REGISTRATION

ANNUAL SIGFORTH CONFERENCE MARCH 7-9, 1991

COMPLETE & MAIL THIS FORM TO:

SIGFORTH CONFERENCE
THE SOFTWARE CONSTRUCTION CO.
2900B LONGMIRE DRIVE
COLLEGE STATION, TX 77845
(409) 696-5432

PLEASE TYPE OR PRINT CLEARLY

NAME ______________________

ORGANIZATION ______________________

ADDRESS ______________________

CITY ______________________ STATE ____

ZIP ________ PHONE ______________

FAX ______________

CONFERENCE FEE:
PAYMENT BY CHECK ONLY PAYABLE TO: SIGFORTH '91

$190	ACM or SIGFORTH MEMBER
$190	GOVERNMENT (AGENCY) ____________
$240	NONMEMBER
$50	STUDENT
$50	LATE FEE (AFTER 12/31/90)

HOTEL REGISTRATION:

HYATT REGENCY OF SAN ANTONIO
123 LOSOYA STREET
SAN ANTONIO, TX 78205
PHONE: (512) 222-1234
FAX: (512) 227-4925

FORTH INTERRUPT HANDLING

BARRIE B. WALDEN - WOODS HOLE, MASSACHUSETTS

Microprocessor designers provide interrupt capabilities as a means for an external event to obtain the central processing unit's attention even while it is busy with another task. The effect is just what the name implies: the receipt of an interrupt causes the processor to temporarily put aside its current task and service the interrupting hardware. To be successful, the following must occur:

1. The microprocessor must receive an indication that an event has occurred requiring special attention. Most microprocessors recognize multiple types of both hardware and software interrupt signals.

2. The microprocessor must stop what it is doing in a manner which allows resuming where it left off after completion of interrupt servicing.

3. A method must exist for determining which special event has occurred out of many possibilities.

4. The microprocessor must have a means for determining the correct action to take in response to each possible interrupt event.

5. There must be a way to resume the original task following completion of the appropriate interrupt response.

Most microprocessors recognize interrupts of various types and priority levels from both hardware and software sources. When an interrupt request is received, the microprocessor stores all of the information necessary to mark its place for later continuation, and then conducts some operation which allows determining the interrupt source and responding in the correct manner. Using the eight-bit 6809 as an example, three hardware and three software interrupt sources are recognized plus hardware reset. All are handled in a similar manner, in that each type has an assigned memory location which must contain the address of the code to be run in response to an interrupt of the associated type. This is a simple arrangement which allows a programmer to easily change the desired responses by changing the appropriate vector, but all interrupts of the same type share a single response vector address. As a result, if multiple sources are available for a particular type of interrupt, the response code must contain a means for differentiating between them. One common method involves using a polling routine which causes the processor to run down a list of possible sources, checking each one for an active indication.

This builds words with a separate code block in the parameter field.

The goal of this article is to show how a small set of Forth words can simplify the use of interrupts by a Forth program. Unfortunately, development of the required words is hardware dependent and therefore, for most readers, the code presented needs to be understood and modified rather than simply copied. Three principal words will be developed: `INTERRUPT`, `ENABLE`, and `DISABLE`. `INTERRUPT` is to be a compiling word which will construct a dictionary entry for each interrupt source. This entry will provide a name for the interrupt and will contain pointers to Forth words for accomplishing the required enable, disable, test, response, and run functions. The run function defines what will happen if the name of the interrupt is encountered in the input stream under a non-interrupt condition.

Once these words have been developed, the process for establishing a working interrupt is as follows:

- Write a Forth word which performs the tasks necessary to enable the interrupt source. This is likely to be a code word which changes values in hardware registers.

- Write a Forth word which disables the interrupt source.

- Write a Forth word which tests the interrupt device—to determine if it is the source of a received interrupt—and returns a flag. This is undoubtedly a code routine, and the flag can be a testable microprocessor condition code, provided that it is not likely to be altered by the code associated with the Forth inner interpreter.

- Write a Forth word which does whatever is desired when the interrupt occurs. This is likely to be a high-level definition.

- Write a Forth word which accomplishes something useful when the name of the interrupt is entered, perhaps initialize a value or output a status message.

- Create the interrupt word by combining information on all of the above words into a single dictionary entry using the compiling word `INTERRUPT`. This might look like the following for a timer interrupt to be named `T1`:

```
INTERRUPT T1
T1_ENABLE
T1_DISABLE
```

```
T1_TEST
T1_RESPONSE
T1_RUN
```

The interrupt can now be enabled and disabled with:
`ENABLE T1` and `DISABLE T1`

When an interrupt occurs, `T1_TEST` will be run and, if the source is the `T1` hardware, `T1_RESPONSE` will be run. If `T1` is encountered in the input stream (i.e., entered from the keyboard), `T1_RUN` will be executed, perhaps to initialize timer number one.

Developing the three principal words is tricky and requires knowledge of both the hardware and the inner workings of the version of Forth being used. Fortunately, the necessary information is usually provided by the hardware and software vendors. `INTERRUPT` is perhaps the most difficult to understand. It is a compiler word which creates a dictionary entry containing pointers to the other words written for this interrupt, as well as some code which can execute these words. The pointers and code are placed in the parameter fields of the words which `INTERRUPT` defines. The code is intended to be run by the microprocessor's primary interrupt-handling routine rather than the normal Forth system. A branch instruction is included which determines if the *response* word is to be executed, depending upon a flag returned by the *test* word. Code following `DOES>` causes the *run* word to execute as the normal Forth run-time activity for the defined word.

Listing One provides an `INTERRUPT` definition for a 6809 system. The double use of `[COMPILE]` is interesting: the intent is to place the code field addresses of the words following `INTERRUPT` at run time in the parameter field of the word being created. The first `[COMPILE]` forces compilation of the second `[COMPILE]` rather than its execution. Therefore, the second `[COMPILE]` will be run when the word `INTERRUPT` executes, forcing compilation of the next word in the input stream (note that the name of the word being created is removed from the stream by `CREATE`). In the example given above, the word `T1` is created and the first pair of `[COMPILES]` causes the first two bytes of `T1`'s parameter field to contain the code address for `T1_ENABLE`. In the same way, the second two bytes point at `T1_DISABLE`.

The fifth and sixth bytes point at the test word and they are followed by an address compiled by `HERE 0F + ,`. This is a pointer to a location 15 bytes further down in the definition. `HERE` provides the address of the pointer and `0F` is added to it as an offset value. The same thing is done for the response word entry. The purpose of these entries will be explained shortly.

Finally, on line nine, we get to some executable code. The version of Forth in this example uses the 6809's Y register as the instruction pointer. Normally, a word is placed in the to-be-run state when the instruction pointer is made to contain the address of a pointer to the word's code field address. This address is usually one of the list of addresses within the parameter field of the calling word. In short, if you wish to run a group of words, make a list of their code field addresses, place the address of the first address in the instruction pointer register, and execute the code for `NEXT`. All the words in the list will be run (including the sub-words that define these words), and then the system will crash when trying to use a pointer beyond the end of the list. `DOCOL` and `;S` prevent this crash when the address list is part of a real colon definition.

Our code will set Y equal to the address of the code field address of the test word and will duplicate the action of `NEXT`. We can ignore the crash potential because we will not complete execution of this code block in the normal Forth manner. After the test word has been run, the instruction pointer will point at the memory location following the one containing the test word's CFA, and the crash will be avoided by ensuring that this address also points to a pointer to executable code. This is the purpose of the offset pointers in lines five and seven, mentioned above. They each point to pseudo code field addresses generated by lines 11 and 15 which, in turn, point to lines 12 and 16, containing executable code.

Lines nine and ten are the hand-assembled version of Figure One.

The only trick is in loading Y correctly. The code shown uses a PC relative offset to point back 13 bytes. The last two lines are the 6809's version of `NEXT`. The amount accomplished by these three lines of code gives some indication of the power of the 6809 instruction set.

Line 12's code is a flag test which determines if the response word should be run. The carry-condition code bit is used for the flag, since it is not affected by the code associated with `NEXT`. In the example, a clear carry flag indicates that the test word did not find the interrupt source, and a branch is used to prevent running the response word. If the carry bit is set, a second manipulation of the Y register occurs to run the response word and return to the code in line 16. Line 16 sets the carry bit to signal successful interrupt servicing to the system's primary interrupt handler. Line 17 contains a return-from-subroutine instruction, which will be explained in a moment.

Line 18 calculates the address of the run word's address by adding a suitable offset to the address placed on the data stack by `DOES>`. This allows executing the run word as the run-time action of the word being created.

The words created by `INTERRUPT` are certainly a little strange. They act like Forth words and they run other Forth words, but their parameter field forms an independent code block containing code field address lists followed by executable code including copies of the code for `NEXT`. This block is called by the microprocessor's interrupt handler and, once running, it becomes a supervisory task overseeing the operation of more normal Forth words. The microprocessor's interrupt handler calls these words by doing a subroutine jump to the executable code in the middle of the parameter field (line nine in the example). This code mimics Forth's normal operation solely because it is an easy way to allow standard Forth words to be run. The code block is not a Forth word, it is not called by a Forth word, and it is not exited in the manner of a Forth word.

`ENABLE` and `DISABLE` are a little more normal. They each have two tasks to accomplish: they must deal with the system's interrupt handling code and they must enable or disable the interrupt source hardware. Continuing with the 6809 example, let us assume that there are multiple sources for the type of interrupt we are interested in. When an interrupt of this type is received, the microprocessor fetches an address from a fixed memory location designated by the manufacturer and begins executing the code at that address. With multiple possible sources, this code must be the source-determination polling routine. Our ENABLE must add to the list of devices polled and our test and response words must return a result flag which the polling routine understands. DISABLE must remove a possible source from the polling list and prevent further interrupts from the hardware. To simplify the example, we will assume that two words exist: IRQV_PUSH and IRQV_PULL add or remove the address on the top of the data stack from the list of address vectors used by the system interrupt handler polling routine. If an address is in the polling routine list, the code at that address will be run as a source-test subroutine. Note that this explains why the code put in place by INTERRUPT ends with a return-from-subroutine instruction.

Listing One shows the code for ENABLE and DISABLE, which are quite similar. Each looks ahead to the next word in the input stream and determines the proper entry point for the polling routine to use in order to cause the test word to run. This entry point is always a fixed distance into the code block contained within each interrupt word's parameter field. The address is placed on the data stack and either IRQV_PUSH or IRQV_PULL is called. Following that, the address of either the *enable* or *disable* word is fetched from the parameter field and the appropriate word is executed. Using our example, ENABLE T1 would result in timer T1 being tested as part of the polling routine (T1_TEST) and timer T1 being configured to generate interrupts (T1_ENABLE). DISABLE T1 would reverse the action. Watch the order in which the two tasks are accomplished by each word—the hardware should never be in the enabled state without the polling address in place.

Consider what we have done: INTERRUPT builds words which, in addition to normal Forth characteristics, contain a separate code block in the parameter field which is run by the system's interrupt handler. This code block acts like the normal Forth inner interpreter and, therefore, can run external high-level Forth words. In addition, the block serves as a storage location for the addresses of other words associated with an interrupt so that words such as ENABLE and DISABLE can find them and execute them. ENABLE and DISABLE interact directly with the system's interrupt handler, accomplishing such tasks as adding and removing addresses from the polling routine list and, additionally, running the Forth words which enable and disable the interrupting hardware.

Listing Two shows how the above can be applied to the Motorola 68HC11 microprocessor. The version of Forth used is that of New Micros, Inc., which meets the Forth-83 Standard and can be purchased permanently installed in the HC11's ROM. The listing includes definitions for a simple interrupt polling routine and the test word address-push and -pull routines. The following comments may help.

New Micros' Forth is configured as follows:

System stack = Return stack
Register Y = Data stack pointer
Addr $0000 = Word pointer (W)
Addr $0002 = Instruction pointer (IP)

INITIALIZE_IRQVT starts a polling address vector table at RAM location $CF00 by setting the first two eight-bit locations to zeroes.

IRQ_POLL is a simple interrupt source polling routine. It is not used as a Forth word; instead, its code address must be placed in the HC11's interrupt vector table so that it will be run when an interrupt of the desired type is received (See SET_RTI_VECTOR in Listing Three). Note that the code ends with a return-from-interrupt instruction rather than the normal JMP NEXT. Both the W and IP values are saved and restored by this routine. Additionally, the data stack pointer (register Y) is set to an unused RAM location ($7000) before jumping to the test routines, to ensure that its use will not cause damage. This is necessary because New Micros' word FIND may be interrupted, and it uses the Y register for more than the data stack pointer.

IRQV_PUSH and IRQV_PULL expect an address on the data stack which they either put in or take out of the polling vector table. For code simplicity, a push can expand the table but a pull cannot shrink it. A $0000 entry marks the end of the table and $FFFF indicates an unused entry location. Note that the last line of code in both of these words requires that you know the address of NEXT.

The definition for INTERRUPT begins with 0 , to put a dummy 16-bit value in the parameter field of the words to be defined. This seems to be required to make the New Micros version of the CREATE ... DOES> construct work properly.

Loading of the IP is done differently than with the 6809 because the HC11 does not have the load-effective-address instruction. Note that the IP is stored in a memory address ($0002) rather than a register.

The New Micros' NEXT code is quite a bit longer than the 6809 version, and it increments the IP value before using it to load register X with a pointer to the code field address (the 6809 post-increments the IP value). The IP and register X already contain the correct values by the time this in-line version of NEXT is reached and, therefore, the normal incrementing and loading is not included.

Listing Three provides some test words which are specific to the New Micros 68HC11, due to the interrupt vector and control register addresses used. They work with the processor's real-time interrupt capability which, once enabled, is made to repeatedly increment the variable CNT.

Barrie B. Walden is the manager of Submersible Engineering and Operations at the Woods Hole Oceanographic Institution, home of the hard-working ALVIN submersible, whose 13,000' depth capability is second in the U.S. only to the Navy's Sea Cliff. He also manages Operational Scientific Services (providing support to scientists on several seagoing facilities) and is one of three principal investigators developing the Autonomous Benthic Explorer (ABE), an unmanned, untethered research vehicle. Ten years ago, he typed in 6800 fig-FORTH and, since then, has spent more time "improving" the language than writing applications. Noting that, for many applications, the source code—even the source language—is never seen by anyone but the original programmer, he relates that the navigation tracking-and-display system used in ALVIN is written in LMI Forth, that ABE's top-level controllers will likely use New Micros' Forth, and that many embedded programs in the ships' data-logging system-interface boxes were developed with Forth.

Figure One.

```
LEAY F3,PCR  (318C F3)    Load IP (Reg Y)
LDX ,Y++     (AEA1)       6809's NEXT
JMP [,X]     (6E94)
```

Listing One. 6809 Interrupt Words.

```
1     : INTERRUPT CREATE
2          [COMPILE] [COMPILE]      \ Enable word
3          [COMPILE] [COMPILE]      \ Disable word
4          [COMPILE] [COMPILE]      \ Test word
5          HERE 0F + ,              \ Pointer to line 11
6          [COMPILE] [COMPILE]      \ Response word
7          HERE 16 + ,              \ Pointer to line 15
8          [COMPILE] [COMPILE]      \ Run word
9          318C , F3 C,             \ Load IP
10         AEA1 , 6E94 ,            \ NEXT
11         HERE 2 + ,               \ Pseudo CFA
12         240B ,                   \ Branch carry clear
13         318C , EC C,             \ Load IP
14         AEA1 , 6E94 ,            \ NEXT
15         HERE 2 + ,               \ Pseudo CFA
16         1A01 ,                   \ Set carry
17         39 C,                    \ Return from subroutine
18         DOES> 0C + @ EXECUTE ;   \ Execute "Run" word


: ENABLE
     BL WORD FIND              \ Find following word's CFA
     DROP DUP                  \ Drop flag, Dup addr
     12 + IRQV_PUSH            \ Load polling vector table
     4 + @ EXECUTE             \ Execute "Enable" word

: DISABLE
     BL WORD FIND              \ Find following word's CFA
     DROP DUP                  \ Drop flag, Dup addr
     6 + @ EXECUTE             \ Execute "Disable" word
     12 + IRQV_PULL            \ Unload polling vector table
```

Listing Two. 68HC11 interrupt words (in New Micros' Forth).

Code words for source polling:

```
CODE INITIALIZE_IRQVT     \ Test routine vector table @ $CF00
  CE C, CF00 ,            \ LDX #RAM_TABLE_ADDR
  6F00 ,                  \ CLR ,X
  6F01 ,                  \ CLR 1,X
  7E C, FE4A ,            \ JMP NEXT
END-CODE
```

(Continued on next page.)

(Continued.)

```
CODE IRQ_POLL
  DE00 ,                  \ LDX $0000 Get W
  3C C,                   \ PSHX Save W
  DE02 ,                  \ LDX $0002 Get IP
  3C C,                   \ PSHX Save IP
  18CE , CF00 ,           \ LDY #RAM_TABLE_ADDR
  CDEE , 00 C,            \ Start LDX ,Y
  2717 ,                  \ BEQ END Table end?
  8C C, FFFF ,            \ CPX #$FFFF Valid entry?
  270C ,                  \ BEQ Int1
  183C ,                  \ PSHY
  18CE , 7000 ,           \ LDY #$7000 Set RP to Ram
  AD00 ,                  \ JSR ,X Jump to test routine
  1838 ,                  \ PULY
  2506 ,                  \ BCS End  Source found
  1808 , 1808 ,           \ Int1 INY INY Point at next entry
  20E4 ,                  \ BRA Start
  38 C,                   \ End PULX Restore IP
  DF02 ,                  \ STX $0002
  38 C,                   \ PULX Restore W
  DF00 ,                  \ STX $0000
  3B C,                   \ RTI * Not a normal Forth return *
END-CODE

CODE IRQV_PUSH
  CDEE , 00 C,            \ LDX ,Y Get addr from data stack
  1808 , 1808 ,           \ INY INY Adj data stack pointer
  183C ,                  \ PSHY
  18CE , CF00 ,           \ LDY #RAM_TABLE_ADDR
  CDAC , 00 C,            \ L1 CPX ,Y
  2717 ,                  \ BEQ L4 if entry already in place
  18EC , 00 C,            \ LDD ,Y Get entry
  270C ,                  \ BEQ L2 Branch if end of table
  1A83 , FFFF ,           \ CPD #$FFFF
  2709 ,                  \ BEQ L3 Branch if inactive entry
  1808 , 1808 ,           \ INY INY Point at next entry
  20EA ,                  \ BRA L1 Loop back
  18ED , 02 C,            \ L2 STD 2,Y Add to table length
  CDEF , 00 C,            \ L3 STX ,Y Add entry
  1838 ,                  \ L4 PULY
  7E C, FE4A ,            \ JMP NEXT
END-CODE

CODE IRQV_PULL
  CDEE , 00 C,            \ LDX ,Y Get addr from data stack
  1808 , 1808 ,           \ INY INY Adj data stack pointer
  183C ,                  \ PSHY Save data stack pointer
  18CE , CF00 ,           \ LDY #RAM_TABLE_ADDR
  CDAC , 00 C,            \ L1 CPX ,Y
  270B ,                  \ BEQ L2 Branch if entry found
  18EC , 00 C,            \ LDD ,Y Get entry
  270C ,                  \ BEQ L3 Branch if end of table
  1808 , 1808 ,           \ INY INY Point at next entry
  20F0 ,                  \ BRA L1 Loop back
  CC C, FFFF ,            \ L2 LDD #$FFFF
  18ED , 00 C,            \ STD ,Y Cancel entry
```

(Continued on page 16.)

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.

(Continued from page 14.)

```
  1838 ,                    \ L3 PULY Restore data stack pointer
  7E C, FE4A ,              \ JMP NEXT
END-CODE
```

68HC11 principal interrupt words:

```
1    : INTERRUPT CREATE
2       0 ,
3       [COMPILE] [COMPILE]    \ Enable word
4       [COMPILE] [COMPILE]    \ Disable word
5       [COMPILE] [COMPILE]    \ Test word
6       HERE 15 + ,            \ Pointer to line 16
7       [COMPILE] [COMPILE]    \ Response word
8       HERE 22 + ,            \ Pointer to line 24
9       [COMPILE] [COMPILE]    \ Run word
10      CE C, HERE 0B - ,      \ Load IP - LDX addr of line 5
11      DF02 ,                 \           STX IP
12      EE00 ,                 \ LDX 0,X  Get code pointer
13      DF00 ,                 \ STX => W
14      EE00 ,                 \ LDX 0,X  Get code address
15      6E00 ,                 \ JMP 0,X  End of NEXT
16      HERE 2 + ,             \ Pseudo CFA
17      2417 ,                 \ Branch if carry flag clear
18      CE C, HERE 18 - ,      \ Load IP - LDX addr of line 7
19      DF02 ,                 \           STX in IP
20      EE00 ,                 \ LDX 0,X  Get code pointer
21      DF00 ,                 \ STX => W
22      EE00 ,                 \ LDX 0,X  Get code address
23      6E00 ,                 \ JMP 0,X  End of NEXT
24      HERE 2 + ,             \ Pseudo CFA
25      0D C,                  \ Set carry flag
26      39 C,                  \ RTS
27   DOES> 0C + @ EXECUTE ;    \ Execute "Run" word
```

```
: ENABLE
   BL WORD FIND          \ Find following word
   DROP DUP              \ Drop flag; Dup addr
   14 + IRQV_PUSH        \ Load addr into polling routine
   6 + @ EXECUTE ;       \ Execute "Enable" word
: DISABLE
   BL WORD FIND
   DROP DUP
   8 + @ EXECUTE         \ Disable hardware
   14 + IRQV_PULL ;      \ Remove addr from polling routine
```

Listing Three. Real-time interrupt test words for New Micros' 68HC11.

```
VARIABLE CNT

: SET_RTI_VECTOR              \ Set RTI interrupt vector to IRQ_POLL
     7E B7E6 EEC!             \ $B7E6 is EEPROM
        [ ' IRQ_POLL @ >< FF AND ] LITERAL B7E7 EEC!
     [ ' IRQ_POLL @ FF AND ] LITERAL B7E8 EEC! ;

CODE CLEAR_CC                 \ Enable Hardware Interrupts
     86 C, 40 C, 06 C,
     7E C, FE4A ,             \ JMP NEXT
END-CODE

: RTI_ENABLE
        40 B024 C!            \ Set enable bit
        CLEAR_CC              \ Enable interrupts
        ." RTI Enabled " ;

: RTI_DISABLE
     00 B024 C!               \ Clear enable bit
     ." RTI Disabled " ;

CODE RTI_TEST
     B6 C, B025 ,             \ Get RTI status
     48 C, 48 C,              \ Bit 6 to carry
     7E C, FE4A ,             \ JMP NEXT
END-CODE

: RTI_RESPONSE
     40 B025 C!               \ Clear flag
     1 CNT +! ;               \ Increment count

: RTI_RUN
     03 B026 C!               \ Set interrupt rate
     0 CNT !                  \ Initialize CNT
     ." RTI Rate Set " ;

INITIALIZE_IRQVT
SET_RTI_VECTOR
INTERRUPT  RTI  RTI_ENABLE  RTI_DISABLE  RTI_TST  RTI_RESP  RTI_RUN
```

(Continued from page 9.)

e. Exception handling
f. Vectored execution
g. Compiler extension operators
h. Data description operators
i. ROM-based applications
j. Any other areas that emerge from the study as representing significant problem areas.

3. Proposed modifications to Forth-83 shall be deemed unacceptable if they result in significant variance from "accepted practice" as identified in Item 1b above, or if the proposed definition is outside the standards of clarity and unambiguity required of an ANS.
4. Once an ANS Forth has been approved, the TC may address proposed standards for language extensions beyond the scope of item 1 above. Areas in which such extensions may be considered include data-base support and graphics. Other extensions will doubtless emerge, and may be considered at the discretion of the TC following approval of ANS Forth.
5. The TC may review existing and proposed standards for other languages.
6. The TC will consider areas in which the BASIS document or accepted practice is in conflict with modern hardware characteristics.
7. The TC will primarily consider one's and two's complement architectures.

INTERRUPT-DRIVEN COMMUNICATIONS

RAMER W. STREED - NORTH MANKATO, MINNESOTA

This article describes a serial communications interrupt handler for the 8250 UART. Its use is shown in a communication program written in HS/FORTH for a PC host to develop Forth applications programs in embedded microprocessors. Forth's interactive nature makes a superior environment for developing programs.

This program was initially written to develop code for a Rockwell 65F12. Simply polling the communications port will not keep up with 2400 baud on a 4.77 MHz 8088.[1] ROM BIOS interrupt calls cannot be used to receive data at rates much faster than 1200 baud. To overcome that limitation, I wrote an interrupt driver for the 8250 UART. Incoming data is placed in a FIFO buffer to be read by the main program. This interrupt handler is the core of the simple terminal program `CO` (short for Comm. On). I am currently using `CO` at 9600 baud to send source code to a Zilog Super 8 running Inner Access F83S8 Forth.

First, I need to describe a few of the differences between HS/FORTH and most 16-bit Forths. HS/FORTH uses a separate segment for code words, for dictionary headers, for parameter and return stacks, and for the body of Forth words. Each segment can be as large as 64K without the overhead of 32-bit addresses. `VAR` is a defining word that creates a data type of variable that returns the value. To get the address, preface the word with

```
THE {var.name}
```

and to store a new value use

```
nnn IS {var.name}
```

Other Forth implementations support similar data types called `QUAN` or `VALUE`. I will enclose words I have defined in braces, since my naming convention uses lowercase. The rest of the code conforms to the Forth-83 Standard.

Communication parameters—baud rate, parity, number of bits, and the number of stop bits—are set by DOS using `MODE`. The word `SHELL` loads a second copy of COMMAND.COM and executes the quoted string. You may set the communication parameters directly, storing the constants in the UART registers.[2]

`{65F12}` and `{S8}` are setup words for the microprocessors I commonly use. They contain the solutions for two of the problems I encountered developing this code. I did not know the turn-around character for the Zilog S8. Thanks to Forth's interactivity, I dumped the serial input buffer to find the answer: an ASCII line feed (0Ah). The other problem surfaced when the program was first started; often I would have to try several times to get `CO` to receive the sign-on message from the Super 8. The UART was not being reset, which effectively disabled the interrupts. The solution is to read the UART and drop the byte to insure the state of the UART on startup.

Add as many bells and whistles as you wish...

The core of the program is the interrupt handler code word `{com.int}`. First, all registers used are pushed on the stack. Next, the data segment is computed from a known location in the code segment (remember, HS/FORTH uses separate segments for code and data) and moved to the DS register. Locating the data segment was one of the more difficult problems I encountered.

Forths that are all in one segment can eliminate the extra step of locating the data segment relative to the code segment. The DX register is loaded with the base address of the communications port. Adding the offset of the Line Status Register (LSR) sets DX for the status read. `AL IN-DX.` reads the LSR into register AL. AL is then checked for errors. The error flag is moved to `{ser.in.error?}`.

Program execution proceeds at the label `{no.error}`. >>> is an HS/FORTH word that defines an assembler label. The code at `no.error` buffers the input. The DX register is reset to point to the communication port base address. The data byte is read from the serial port. The data is then stored in the input buffer at the offset of `{ser.in.head}`. `{ser.in.head}` is incremented and wrapped to form a circular input buffer. Wrapping the input buffer with a logical `AND` requires the buffer size to be a power of two but is fast, so it is ideal for interrupt routines which must be kept short. Next the buffer pointers are checked to be sure that the input buffer was not overrun. If there is no overrun, the new pointer offset is stored at `{ser.in.head}`.

The exit code is at `EOI`. The interrupts are disabled, as there is no need to stack pending interrupts now. The 8259 interrupt controller is reset, and the stack restored. The `IRET` instruction enables the interrupts so that the process can be repeated.

Two error conditions are checked: serial input errors from the Line Status Register and buffer overflow. The LSR register error bits are isolated and returned in `{ser.in.error?}`. Any non-zero value is an error. The high-level Forth program can analyze the returned value to give more meaningful error messages. The buffer overflow is indicated in `{buffer.overflow?}`; a false is no overflow, true is overflow condition. The error flags are not reset by the interrupt handler. It is up to the program to clear them after the error is acknowledged.

After the interrupt handler is compiled, it must be installed so that the 8259 interrupt controller can call the correct routine. This is done with {IV=com1}. INT-VECTOR is a defining word that sets up the address array so that the interrupt vector can be pointed to {com.int}. This is done later with {IV=com1} INSTALL.

To prevent executing {com.int} (it is not an executable word), it is beheaded. Beheading removes {com.int} from the dictionary so it cannot be found. BYE is redefined to REMOVE the interrupt vector before exiting Forth, since exiting would eliminate the code that serviced the interrupt.

The next words initialize the communications hardware. The IBM PC-specific addresses are in constants at the beginning of the source code to make changes for different hardware easier. As coded, the program runs only on COM1. However, I have made all communication address variables, so the code could be extended to run for any communications port. {serial.int.enable} sets up the UART and 8259 interrupt controller; likewise, {serial.int.disable} clears the UART and 8259.

{serial.out.empty?} checks to see if the UART can accept a data byte for transmission. It is used to define {cemit} which works just like the system CEMIT and emits a byte to the communications port. {serial.in.buffer@} gets a byte from the input buffer to the stack. {serial.input?} returns a true flag if a data byte is available. They are used to define {ckey} which works just like the system CKEY except the input is from the interrupt-driven input buffer and not a DOS call. *Note:* these words directly control the hardware—they do not use BIOS interrupt 14h.

{serial.on} and {serial.off} install or remove the interrupt vector, enable or disable the UART and 8259 interrupt controller, and set a flag that indicates the status of the serial port.

{poll.keyboard} has three functions. First, it tests for a control-Z and leaves a true flag to exit. Second, it tests for a backspace (08h) and converts it if found to a rubout (7Fh). This was necessary for the Rockwell 65F12, which objected to the backspace. The Zilog Super 8 accepts either backspace or rubout, so I left this in all the time. Other special character conversions could be implemented here if required. Third, the character is sent to the communications port.

{poll.ser.input} tests for serial input. Carriage returns and line feeds are checked and handled as special cases to ensure correct results. If the log file has been opened, it sends the input to the log file. All input is sent to the screen via SEMIT.

The main routine CO is a simple loop that repeatedly checks the keyboard, errors, and serial input. This loop can have as many bells and whistles added as you wish, since the interrupt-driven input buffer will take care of input while the computer is busy checking function keys or other extensions.

In my work developing embedded application programs on other microprocessors, the key flexibility offered by Forth is the ability to write the code on a PC using my text editor and the computer's disk storage. This requires a few more words to send the files to the target system. The words are: ?emit, CSEND, $SEND, LSEND, SEND, and SAY. LSEND looks for the turn-around character after a line is sent from the host to the target, so the target has time to process the data before more is sent. The turn-around character is a null for the Rockwell 65F12 and a line feed 0Ah for the Zilog Super 8. SAY sends the following string up to the carriage return to the target computer and waits for reply. SEND sends a complete file to the target computer, as in SEND filename. Both SAY and SEND echo the data to the screen if EMIT? is true. This function is handled by {?emit}.

Many features could be added to this program. The interrupt handler could have DTR, RTS, or XON/XOFF software flow control added to stop the other computer from sending more data if the input buffer filled up. File upload and download with XMODEM or Kermit could be added for use in on-line communications.

I would like to see a full communications program developed and added to the Forth Model Library. Will one of the readers please write a series of articles on communications, like Al Stevens' series constructing SMALLCOM (see references).

My thanks to Mahlon Kelly and Donald P. Madson for their assistance with the code for this program.

References
1. Teza, Jeffrey R. "Multitasking Modem Package," *Forth Dimensions*, Volume IX, Number 6.
2. Fox, Brian. "8250 UART Revisited," *Forth Dimensions*, Volume XI, Number 1.
3. Cooper, Paul. "Menu-Driving the 8250 Async Chip," *Forth Dimensions*, Volume X, Number 4.
4. Stevens, Al. "TINYCOMM: The Tiny Communications Program," *Dr. Dobb's Journal*, February 1989.
5. Stevens, Al. "TINYCOMM Begets SMALLCOM," *Dr. Dobb's Journal*, March 1989.

```
\ interrupt driven serial port words
\ Ramer W. Streed
\ 474 Marvin Blvd.
\ North Mankato, Minnesota 56001

\ Check for definitions of pc@ and pc! if not available define
\ p@ = word (16 bit) fetch,  pc@ = byte (8 bit) fetch
FIND79 pc@  0= ?( SYNONYM pc@ P@ SYNONYM p@ PW@ SYNONYM pc! P! SYNONYM
p! PW! )

VOCABULARY TERMINAL
ONLY FORTH ALSO TERMINAL ALSO
TERMINAL DEFINITIONS

\ ibm pc com port specific words

HEX
 03F8 CONSTANT com1.base                          \ first port adr for com1
 0010 CONSTANT com1.int.mask                      \ irq 4
 000C CONSTANT com1.int#                          \ interupt vector number
```

```
   20 CONSTANT 8259.eoi                     \ command to reset 8259
   20 CONSTANT 8259.base                    \ first port adr for 8259 pic
 1000 CONSTANT ser.in.buffer.size           \ 4096 byte ring buffer
CREATE  ser.in.buffer  ser.in.buffer.size ALLOT
    0 VAR ser.in.head                       \ insert new bytes at head pointer
    0 VAR ser.in.tail                       \ remove bytes at tail pointer
    0 VAR ser.in.error?                     \ non zero if serial error occurs
    0 VAR buffer.overflow?
FALSE VAR serial.on?                        \ True if Com is enabled
   0A VAR turnaround                        \ Turnaround Character 0 or 0A
com1.base VAR com.base                      \ pass com port base address to IV
com1.int.mask VAR com.int.mask              \ pass intrrupt mask to routine
com1.int# VAR com.int#                      \ pass interrupt number
  01E CONSTANT ser.error.mask               \ mask non error bits
    1 CONSTANT IER                          \ Interrupt Enable Register offset
    3 CONSTANT LCR                          \ Line Control Register offset
    4 CONSTANT MCR                          \ Modem Control Register offset
    5 CONSTANT LSR                          \ Line Status Register offset
    6 CONSTANT MSR                          \ Modem Status Register offset

: 2400BAUD $" MODE COM1:2400,N,7,2" SHELL MAIN CR ;
: 9600BAUD $" MODE COM1:9600,N,8,1" SHELL MAIN CR ;

: 65F12  0 IS turnaround 2400BAUD com.base pc@ DROP ;  \ reading UART resets
: S8    0A IS turnaround 9600BAUD com.base pc@ DROP ;  \ data ready interrupt


\  Serial Communications Interupt Handler

CODE com.int
       STI.                            \ sti = enable ints
\      NOP.                            \ patch to 0CC to enter Debug
       AX PUSH.                        \ save regs used
       BX PUSH.
       DX PUSH.
       DS PUSH.
       AX CS MOV.                      \ set up data segnment
       CS: AX CA' EXIT 2- +[] ADD.     \ data segment offset from CS
       DS AX MOV.
       DX THE com.base +[] MOV.
       DX LSR IW ADD.                  \ Line Status Register address
       AL IN-DX.                       \ get error status
       AX ser.error.mask IW AND.       \ mask non error bits
       JZ no.error
       THE ser.in.error? +[] AX MOV.   \ save error flag
>>> no.error
       DX LSR IW SUB.                  \ receiver buffer register
       AL IN-DX.                       \ get byte from serial port
       BX THE ser.in.head +[] MOV.     \ buffer pointer offset to BX
       ser.in.buffer +[BX] AL MOV.     \ move byte to buffer address + offset
       BX INC.                         \ adjust pointer for new character
       BX ser.in.buffer.size 1- IW AND. \ wrap pointer
       THE ser.in.tail +[] BX CMP.     \ heads must not catch up to tails
       JE overflow
       THE ser.in.head +[] BX MOV.     \ save new pointer offset
>>> EOI
       CLI.                            \ disable interrupts
       AL 8259.eoi IB MOV.
       8259.base AL OUT.               \ reset 8259 int
       DS POP.                         \ Restore stack
       DX POP.
       BX POP.
       AX POP.
       IRET.                           \ Return from Interrupt & Enable interrupts
>>> overflow
       THE buffer.overflow? +[] FF IW MOV.  \ overflow flag to Forth variable
```

```
      JMPS EOI
      ?CSP                                  \ Insure Forth stack compiled correctly

ONLY FORTH ALSO TERMINAL ALSO               \ Be sure the definitions go in the
TERMINAL DEFINITIONS                        \ correct vocabulary

\ patch ibm pc com1 interrupt vector

com1.int# INT-VECTOR IV=com1 CA' com.int IV=com1 !

BEHEAD' com.int                             \ prevents com.int from being executed

: BYE  IV=com1 REMOVE  BYE ;                \ insures interupt handler is removed



\ initialize ibm pc com hardware

: serial.int.enable
    com.base LCR + DUP                      \ Line Control Register LCR
    pc@ 7F AND SWAP pc!                     \ div latch access off
    1 com.base IER + pc!                    \ int enable register
    0B com.base MCR + pc!                   \ modem control dtr & out2
    [ 8259.base 1+ ] LITERAL DUP pc@        \ read 8259 int mask
    com.int.mask -1 XOR AND SWAP            \ turn on desired level
    pc!                                     \ write new mask to 8259
    0 IS ser.in.head  0 IS ser.in.tail      \ init ring buffer ptrs
    ser.in.buffer ser.in.buffer.size ERASE ;  \ Clear ring buffer

\ disable ibm pc com interrupt

: serial.int.disable
    [ 8259.base 1+ ] LITERAL DUP
    pc@                                     \ read 8259 interrupt mask
    com.int.mask OR SWAP                    \ turn off this interrupt
    pc!                                     \ write mask back to 8259
    com.base LCR + DUP                      \ Line Control Register LCR
    pc@ 7F AND SWAP pc!                     \ div latch access off DLAB=0
    0 com.base IER + pc! ;                  \ disable interrupts int enable register IER

: serial.out.empty?   ( -- f )              \ true if ok to send byte
     com.base LSR + pc@ 20 AND ;

: cemit  ( b --- )
    BEGIN
       serial.out.empty?                    \ wait for transmit ready
    UNTIL
    com.base pc! ;                          \ send one byte to serial port

: ser.in.buffer@   ( --- b )                \ gets byte from ring buffer to stack
    ser.in.buffer  ser.in.tail  IC@         \ Get byte if available
    ser.in.tail  1+                         \ adjust tail pointer
    [ ser.in.buffer.size 1- ] LITERAL AND   \ wrap tail pointer
    IS ser.in.tail  ;                       \ save tail pointer

: serial.input?
    ser.in.tail  ser.in.head  <> ;          \ True if a character is available

: ckey   ( --- c )
    BEGIN
       serial.input?                        \ False if buffer is empty
    UNTIL                                   \ wait for character
    ser.in.buffer@  ;                       \ get byte from ring buffer

\ serial port control words
```

```
: serial.on    IV=com1 INSTALL  serial.int.enable TRUE IS serial.on? ;
: serial.off  serial.int.disable  IV=com1 REMOVE FALSE IS serial.on? ;


: poll.keyboard     ( --- f )              \ ^Z leaves true flag to exit
   ?TERMINAL  DUP
   IF S->C DUP 1A =                         \ test for ^Z
      IF DROP TRUE ELSE DUP 08 =            \ convert backspace to rubout
         IF DROP 7F THEN                    \ Rockwell objects to backspace
      cemit FALSE                           \ false flag since there was a character
      THEN
   THEN ;

FALSE VAR ?LOG
\ OPEN-LOG opens a log file of the interaction. Syntax: OPEN-LOG LOGFILE.LOG
: OPEN-LOG 20 TEXT PAD MAKE-OUTPUT TRUE IS ?LOG ;
\ Closes the log file. Syntax: CLOSE-LOG
: CLOSE-LOG CLOSE-OUTPUT FALSE IS ?LOG ;
: ?FEMIT ?LOG IF FEMIT ELSE DROP THEN ;

: poll.ser.input     ( --- )                \ read input buffer to screen
   serial.input?                            \ is a character available?
   IF ser.in.buffer@ DUP 0A -               \ get character & filter linefeeds
      IF DUP ?FEMIT DUP 0D =                \ send to LOGFILE IF ?LOG is TRUE
         IF 0A ?FEMIT THEN                  \ if character is CR add LF to LOGFILE
         DUP SEMIT 0D =                     \ send to screen
         IF 0A SEMIT THEN                   \ if character is CR add LF to screen
      ELSE DROP THEN
   THEN  ;

: serial.error                              \ Test for serial communications error
   ser.in.error?
   IF
      CR ." Serial Input Error" CR
      FALSE IS ser.in.error?
   THEN ;

: buffer.overflow                           \ Test for serial Buffer overflow
   buffer.overflow?
   IF
      CR ." Serial Input Buffer Overflow" CR
      FALSE IS buffer.overflow?
   THEN ;

: CO ( com-on )                             \ Simple communications program
   CR
   serial.on? NOT IF serial.on THEN
   BEGIN
      poll.keyboard                         \ Test for data to send or quit ^Z
   0= WHILE
      serial.error                          \ Test for serial receive errors
      buffer.overflow                       \ Test for buffer overflow
      poll.ser.input                        \ Read input data if any is available
   REPEAT   MAIN  CR ;                      \ in input buffer


TRUE VAR EMIT?

: ?emit   EMIT? IF SEMIT ELSE DROP THEN ;    \ emits if EMIT? is true

: CSEND ( c -- ) cemit ckey serial.error buffer.overflow ?emit ;

: $SEND ( $addr -- ) COUNT DUP
        IF 0 DO DUP I IC@ CSEND LOOP DROP ELSE DDROP THEN ;
```

```
: LSEND ( $addr -- ) DUP C@
       IF $SEND 0D cemit
          BEGIN
            ckey DUP ?emit turnaround =
          UNTIL
       ELSE DROP THEN ;

: SEND   serial.on? NOT IF serial.on THEN
         20 TEXT  PAD  OPEN-INPUT <FILE CR
         BEGIN
           255 GW LSEND THE R# 0=
         UNTIL
         CLOSE-INPUT MAIN CR ." Done " ;

: SAY   serial.on? NOT IF serial.on THEN
        0D TEXT PAD LSEND MAIN CR ;

DECIMAL EXIT

\ LSEND  looks for turnaround character 0AH or NULL (S8 or R65f12 respectively).

\ SEND <filename> (in a $address)

\ CO = COMON  Terminal conversation with remote computer <Ctrl-Z> terminates
\            communications.

\ SAY   Sends command up to <CR> to remote computer and waits for reply.
\       SAY "command" will send command to remote.
\
\   com1.int.mask is used to set up 8259 interrupt controller for hardware
\   interupt number 4.
\   7 6 5 4 3 2 1 0 - timer                iv 8   adr 20
\   | | | | | | | +---- keyboard           iv 9   adr 24
\   | | | | | | +------ reserved           iv a   adr 28
\   | | | | +-------- com2, sdlc, bsc2     iv b   adr 2c
\   | | | +---------- com1, sdlc, bsc1     iv c   adr 30
\   | | +------------ fixed disk           iv d   adr 34
\   | +-------------- floppy disk          iv e   adr 38
\   +---------------- printer              iv f   adr 3c
\   nmi ------------- parity               iv 2   adr  8
\
\   com1.int# 000C hex = 00001100 binary
\

\   Interrupt Enable Register    com.base 1+
\   high enables interrupt
\
\   3 2 1 0 - receive
\   | | +---- TX empty
\   | + ----- Receive Line Status
\   +-------- Modem Status
\
\   Line Control Register   LCR  com.base 3 +
\
\   7 6 5 4 3 2 1 0 -Word Length Select Bit 0 (WLS0)
\   | | | | | | | +----Word Length Select Bit 1 (WLS1)
\   | | | | | | +------Number of Stop Bits (STB)
\   | | | | | +--------Parity Enable (PEN)
\   | | | +----------Even Party Select (EPS)
\   | | +------------Stick Parity
\   | +--------------Set Break
\   +----------------Divisor Latch access Bit (DLAB)
\
\   Modem Control Register   com.base 4 +
```

(Continued on page 33.)

EAR TRAINING

GLEN B. HAYDON - LA HONDA, CALIFORNIA

Ear training is an important part of a musical education. It is different from music appreciation and playing instrumental music. Hearing has an analogy with vision. Most of us are sighted and we learn to appreciate many visual details. We usually suppress the auditory details except for those associated with the spoken language. Sometimes we do not pay much attention to even the spoken language. We hear only what we want to hear.

The Problem

There are many characteristics of music, including the pitch, the intervals between musical tones, and the types of musical triads. We learn to recognize these characteristics in ear training. We do not need to recognize the differences between major, minor, augmented, and diminished triads to appreciate music. But to develop a better appreciation and understanding of music, we can benefit from a course in ear training.

Well, my wife—after several years of piano—decided to take some additional courses in music. Among those courses, she had to go to a computer laboratory for ear training. There, students used programs running on Apple computers to provide the ear-training drills, including pitches, intervals, triads, scales, and so on. It would be good to extend this practice at home. I use IBM clones. Evidently, the computer laboratory did not have similar programs to run on the IBM family of computers. I made the rounds of several computer software stores and could not find any similar programs. So what was I to do?

The Conceptual Solution

Within a half hour, I had my IBM clone playing individual notes. The rest was just a matter of developing a program that would provide the ear-training exercises. After nearly 20 years of my wife's being a computer widow, she is taking an interest in computers.

The problem is really simple. The IBM clones can set the pitch and turn the speaker on and off. The value 0B6h sent to port 43 allows a calculated 16-bit divisor to be sent to the 8-bit port 42 as two eight-bit values, the low value first. Then, toggling a bit in port 61 turns the speaker on and off. An appropriate delay time can be used.

Program Utilities

The first function to be defined is a delay. A do-nothing loop will serve the purpose. The duration will depend upon the speed of the processor. The easiest way is to make a simple loop and time it with your watch. You can adjust the number of iterations to give a sufficiently accurate measure of the time, in milliseconds or any other unit. This is the only value that you will have to adjust for this program to run on an IBM clone system at a different speed.

We must convert a pitch to a divisor value...

```
{
: MSEC
  0 DO
    9 0 DO LOOP
  LOOP ;
}
```

Changing the value 9 in the source and recompiling the program is probably more difficult than changing the value of DURATION in the compiled program where it is used. This can be done from the run-time program. The name MSEC will no longer produce milliseconds, but the result can be used the same way.

For ear training, it would be desirable to present the training examples in random order. I stole the code for RND, RANDOM, and CHOOSE from Leo Brodie's *Starting Forth*. It is a classical reference and adequate for this purpose.

```
{
CREATE RND  HERE ,

: RANDOM
  RND @ 31421 *
  6927 + DUP
  RND ! ;

: CHOOSE   ( u1 -- u2 )
  RANDOM  U*
  SWAP DROP ;
}
```

The items to be chosen are placed in arrays later in the program.

A formatting tool allows multiple carriage returns.

```
{
: CRS
  0 DO
  CR LOOP ;
}
```

We would also like to position the responses at the same place on the screen each time .

```
{
: POSITION
  PAGE
  10 CRS  15 SPACES ;
}
```

Primitives

The primitive function we need

will take as parameters a pitch in hertz and the duration in milliseconds. As described above, we need to convert the desired pitch to the required divisor value for the 8253 chip. The required value can be calculated from a system-dependent constant divided by the desired pitch. That constant is a double-precision value. I guess it could be calculated easily enough, but it is far easier to find the correct value by experiment. The value can be tuned with a tuning fork or with a tuned piano. If you find the value I have used to be off for your ear, just tune it yourself. The calculated value is stored in the appropriate channel of the 8253 through ports 43 and 42, as described. Once the value is latched into the register, you need to turn the speaker on for the desired delay and then turn the speaker off through port 61. For more details of the algorithm, see any manual on the IBM PC.

```
{
HEX
: NOTE  ( pitch, msec -- )
  B6 43 P!
  11C5F1. 4 ROLL  U/MOD
  SWAP DROP
  DUP FF AND 42 P!
  100 /  42 P!
  61 P@   DUP 03 OR
  61 P!   SWAP
  MSEC 61 P! ;
DECIMAL
}
```

Try this function with 256 for middle C, and 10000 for 10 seconds. The double-precision value can be adjusted for tuning, and the time used will show you how different your system is from mine.

A Tempered Scale

The next problem is to set up the pitches for the 12 notes of the chromatic scale. For each musical key, the diatonic frequencies vary a little with each note. As with a piano, the scale can be tempered. The easiest way is to find a music book that discusses the tempering of the scale. This I have done and assigned the appropriate pitch to each name.

```
{
256 CONSTANT C
271 CONSTANT C#
287 CONSTANT D
304 CONSTANT D#
323 CONSTANT E
342 CONSTANT F
362 CONSTANT F#
384 CONSTANT G
406 CONSTANT G#
431 CONSTANT A
456 CONSTANT A#
483 CONSTANT B
}
```

Next we want to play the notes. For this we first define a `DURATION`.

```
{
CREATE
DURATION 4000 ,
}
```

On my system this is approximately four seconds. The value can be easily modified from the run-time program:

```
1000 DURATION !
```

The new duration will be one quarter as long.

Next, we will eventually want to change the keys for our exercises. We do this by defining a variable that can be changed during the program.

```
{
CREATE KEYS 256 ,
}
```

Then we want to play a note of a selected pitch on the computer. The `DURATION` of each note will be the same.

```
{
: PLAY   ( pitch )
  KEYS @  256 */
  DURATION @ NOTE ;
}
```

Finally, we can define the 12 notes for the middle octave. By changing the value in `KEYS`, the tempered scale can be placed anywhere.

```
{
: C1    C    PLAY ;
: C#1   C#   PLAY ;
: D1    D    PLAY ;
: D#1   D#   PLAY ;
: E1    E    PLAY ;
: F1    F    PLAY ;
: F#1   F#   PLAY ;
: G1    G    PLAY ;
: G#1   G#   PLAY ;
: A1    A    PLAY ;
: A#1   A#   PLAY ;
: B1    B    PLAY ;
: C2    C 2* PLAY ;
}
```

A flat symbol is not available on my computer but a sharp symbol is. In a tempered scale, a C sharp has the same pitch as a D flat. That problem is easily solved. The twelve notes of a chromatic scale beginning with middle C are given a postscript of 1, indicating the middle octave. Adjust the duration of the note to suit your problem.

The Exercises

The next goal is to develop the ear-training exercises.

The first exercise will be learning to recognize a series of triads: major, minor, augmented, and diminished. Assign the appropriate pitches to functions with the appropriate name.

```
{
: MAJOR
  C1 E1 G1 E1 C1 ;

: MINOR
  C1 D#1 G1 D#1 C1 ;

: DIMINISHED
  C1 D#1 F#1 D#1 C1 ;

: AUGMENTED
  C1 E1 G#1 E1 C1 ;
}
```

Each name will execute the selected triad in the selected key, and with the selected `DURATION` for each note.

The code fields of the names of the triads are stored in an array, from which they can be randomly chosen.

```
{
CREATE triads
  ' MAJOR CFA ,
  ' MINOR CFA ,
  ' DIMINISHED CFA ,
  ' AUGMENTED CFA ,
}
```

The program can then choose them randomly from that array. Also, we will want to select the keys at random. Therefore, we define an array of keys. Then we can choose one randomly or we could select any one key from the array.

```
{
CREATE keys
  ' C CFA ,
  ' C# CFA ,
  ' D CFA ,
  ' D# CFA ,
  ' E CFA ,
  ' F CFA ,
  ' F# CFA ,
  ' G CFA ,
  ' G# CFA ,
  ' A CFA ,
  ' A# CFA ,
  ' B CFA ,
}
```

Next we will put together an exercise that will randomly select a key, one of the triads, and play it. A pause after playing the triad allows the user to decide which key and triad were played. Striking any key will then display the answer, followed by another pause to let the user think about his work. Pressing any key will terminate the pause and continue the program.

```
{
: TRIAD
  POSITION
  12 CHOOSE
    2* keys + @
    DUP EXECUTE
    KEYS !
  4 CHOOSE
    2 * triads + @
    DUP EXECUTE
  BEGIN ?TERMINAL UNTIL
  POSITION
  SWAP ." Key of "
    2+ NFA ID.
    2+ NFA ID.
    ." triad. " CR
  BEGIN ?TERMINAL UNTIL ;
}
```

CHOOSE first selects the key and then the triad to be played. The program must remember both the selected musical key and the triad, so that the correct answer can be displayed when ready. Next we can write a function, TRIADS, to repeat the function TRIAD.

```
{
: TRIADS
  CR 100 0 DO
    TRIAD
    ?TERMINAL
    IF LEAVE THEN
  LOOP ;
}
```

You can terminate the exercise by holding down any key until the menu appears. This is done by the conditional leave construct.

Intervals

A similar pattern can be used for a variety of other ear-training exercises. The laboratory my wife used included intervals. For this, the series of intervals is defined with the appropriate names.

```
{
: Perfect_unison  C1 C1 ;
: Minor_second  C1 C#1 ;
: Major_second C1 D1 ;
: Minor_third C1 D#1 ;
: Major_third C1 E1 ;
: Perfect_fourth C1 F1 ;
: Augmented_fourth C1 F#1 ;
: Perfect_fifth C1 G1 ;
: Minor_sixth C1 G#1 ;
: Major_sixth C1 A1 ;
: Minor_seventh C1 A#1 ;
: Major_seventh  C1 B1 ;
: Perfect_octave  C1 C2 ;
}
```

The code fields of these names are placed in an array from which they can be randomly chosen.

```
{
CREATE intervals
  ' Perfect_unison CFA ,
  ' Minor_second CFA ,
  ' Major_second   CFA ,
  ' Minor_third CFA ,
  ' Major_third CFA ,
  ' Perfect_fourth CFA ,
  ' Augmented_fourth CFA ,
  ' Perfect_fifth  CFA ,
  ' Minor_sixth CFA ,
  ' Major_sixth CFA ,
  ' Minor_seventh  CFA ,
  ' Major_seventh CFA ,
  ' Perfect_octave CFA ,
}
```

To start with, it is easier to learn the intervals in only one key. We will start with the key of C. The key of C is the first item, beginning with 0, in the array of keys. The form is maintained for later use in selecting random keys.

```
{
: INTERVAL
  POSITION
  0  2* keys + @
  DUP EXECUTE  KEYS !
  13 CHOOSE
    2* intervals + @
    DUP EXECUTE
  BEGIN ?TERMINAL UNTIL
  POSITION
  SWAP ." Key of "
    2+ NFA ID.
    2+ NFA ID.
    ."  Interval. " CR
  BEGIN ?TERMINAL UNTIL ;
}
```

Once the function for a single INTERVAL is correct, we can define multiple INTERVALS.

```
{
: INTERVALS
  CR 100 0 DO
    PAGE 12 CRS INTERVAL
    ?TERMINAL
    IF LEAVE THEN
  LOOP ;
}
```

Mixed Keys and Intervals

Later, using a similar function, we can randomly select first the musical key and then the interval.

```
{
: KEY&INTERVAL
  POSITION
  12 CHOOSE
    2* keys + @
    DUP EXECUTE   KEYS !
  13 CHOOSE
    2 * intervals + @
    DUP EXECUTE
  BEGIN ?TERMINAL UNTIL
  POSITION
  SWAP ." Key of "
    2+ NFA ID.
    2+ NFA ID.
    ."  Interval. " CR
  BEGIN ?TERMINAL UNTIL ;
}
```

Then we can execute the single function repeatedly.

```
{
: KEY&INTERVALS
  CR 100 0 DO
```

```
    KEY&INTERVAL
    ?TERMINAL
    IF LEAVE THEN
  LOOP ;
}
```

These examples provide a few exercises for drill on the basics of an ear-training program. The exercises can be extended to scales and more complicated chords. That is for the programmer to play with. These will give the beginner to ear training a good start.

Menu

There is another problem for the music student who has never worked with a computer and does not know how to type. To solve that, the exercise options can be put into a simple menu. The menu could be patched into the program so that it is displayed at the start, making it a turnkey program.

```
{
: MENU
  PAGE 4 CRS 10 SPACES
  ." A    TRIADS "
     2 CRS 10 SPACES
  ." B    INTERVALS "
     2 CRS 10 SPACES
  ." C    KEY&INTERVALS "
     2 CRS 10 SPACES
  ." D    QUIT "
     3 CRS 10 SPACES

  ." Enter the letter for "
  ." the drill of choice."
     5 CRS ;
}
```

Each letter is defined as an alias for the desired exercise. Adding MENU to the alias will return the user to the menu.

```
{
: A
  TRIADS  MENU ;

: B
  INTERVALS MENU ;

: C
  KEY&INTERVALS MENU ;

: D
  PAGE  BYE ;
}
```

After the menu is displayed, you are actually in the programming language and can execute any of the available language functions as well as the particular exercises. At this point, the value for DURATION can be adjusted.

Compile this paper and the program will run!

A Postscript

I developed this source code as any programmer might. It required the definition of the problem, a programmer, a writer, an editor, a graphics artist, a printer, and a good proofreader. After I decided what I wanted, I first wrote the program as I do most of mine. I started with Forth screens and left them a mess. After everything was working about the way I had in mind, I reformatted the screens, set off portions with plenty of white space, and often chose more descriptive names. I then took my small portable computer to bed and drafted the text, which I later uploaded to my main system. Then I converted the revised Forth screens to a text file. With my word processor, I merged my text and the source code. Along the way, I edited and reedited the text many times. I used my spelling and grammar programs to review the text. Finally, I moved the edited text file to my typesetting program, where I formatted the output and then printed it. Finally, there were several cycles of proofs and corrections before printing the publication copy.

Conclusion

This paper is actually the source code for my musical ear-training program. The text file compiles. It is written for the reader. The compiler selects only those portions enclosed in braces for compilation. *[To learn more about this uncommon feature, developed to encourage good in-line source documentation, see the author's original discussion, "Formatting Source Code" in* FD *X/6.]*

Any project requires a report. Why not make the report the source code?

Glen B. Haydon is the president of Epsilon Lyra and WISC Technologies, the author of All About Forth *(3rd ed.), and the developer of the widely distributed MVP-FORTH. This paper was presented at the 1990 FORML Conference and is reprinted with permission.*

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

Part Four

FORST: A 68000 NATIVE-CODE FORTH

JOHN REDMOND - SYDNEY, AUSTRALIA

It is a commonplace that good assembly code is tighter and faster than code generated by a high-level language, largely because the hardware registers of the CPU can be used to hold intermediate results and because register operations are much more time and space efficient than those which access memory. A major disadvantage, however, of assembly coding is the need to keep accurate track of the contents of the registers—and to save and restore them as necessary.

A natural development of the use of named local variables is to refine them as register variables. These are formally available in the C language and can significantly improve performance, but the programmer is forced to leave the details of the allocation to the compiler.

This article describes an approach to providing register variables for ForST which adds significantly to the quality of the compiled code. Using slightly modified code which uses a `BEGIN ... UNTIL` inner loop, the Atari ST executes 100 iterations of the infamous Sieve in under 27 seconds. As a more important example, the floating-point code previously described for ForST is twice as fast (only about four times slower than 32-bit integer arithmetic for multiplication and division) and more than ten percent smaller when compiled for register, rather than local, variables.

In the spirit of Forth, a great deal of flexibility is left to the programmer in deciding the precise allocation of the registers. The technique is general, but will obviously be most successful when implemented for a microprocessor with a large number of registers.

Register Allocation in ForST

The 68000 microprocessor has eight data registers (D0–D7) and eight address registers (A0–A7) available to the programmer. Many of these have specific uses in ForST (Figure One).

Using Register Variables

Register arguments and register variables are declared, in the same way as previously described for (local) arguments and local variables, at the start of a word definition. As only six user registers are available, it is permissible to use both register and local variables together in one definition. The one constraint is that `ARGS` and `REGARGS` (in either order) be declared before `REGS` and `LOCALS`.

As indicated in Figure One, the register allocation is in a fixed order. As with ForST locals, a temporary header is set up for each of the named variables. When code generation is started, after the } word is encountered in the definition, the first instruction to be compiled is a save of all the allocated registers to the hardware stack by a multi-register push (using the MOVEM.L instruction) and then the register arguments are popped by a single instruction in the same way. If `ARGS` or `LOCALS` are also used, code is then compiled to set up the stack frame and, if required, to load the args into it. The source-level formalisms are illustrated by a definition of `CMOVE` (Figure Two-a). The code generated (Figure Two-b) is functionally equivalent to, and much faster than, that produced by standard Forth.

Forth's true primitives are not the well-known stack-juggling words...

This is the simplest use of register variables, but there is room for improvement. The order of register allocation (Figure One) indicates that source, destination, and length will be assigned to A3, A2, and D5, respectively. In other words, two address registers are used to point to the source and destination strings. Now, the 68000 permits predecrement and postincrement addressing without any size or speed penalty. Therefore, any code such as that in `CMOVE`, which includes extra instructions to increase the pointers, is not making proper use of the hardware. With a slightly smarter compiler, we can write a shorter definition (Figure Two-c), using an INC flag to direct postincrements after the fetch and store.

Both the source and object code can be improved further by the use of a `FOR ... NEXT` loop. `FOR`, as normally used, expects a number on the stack. When local variables are available, however, it is more flexible and efficient to use it as a prefix for the variable. (The ForST local prefixes are summarized in Figure Four.) `FOR` simply specifies which variable should be decremented when the `NEXT` value is encountered. `CMOVE` can now be coded more cleanly (Figure Two-d). The object code generated from this definition is quite close to what an assembly programmer would write (Figure Two-e).

The (in this case) only really wasted instruction clears register D0 (four clock cycles out of 38 inside the loop). The very best code, using a memory-to-memory byte move, would require 30 cycles in the loop and four bytes less space. A more eccentric example will generate code as good as that from an assembler (Figure Two-f). This word will expand an array of bytes into an array of words (cells). (It really is necessary to do this to output text in the GEM windows environment!) The clearing of the accumulator is necessary in this case to provide blank padding in the word array.

This level of optimization was possible because of the redefinition of a set of smart fetch-and-store (immediate) words, which are able to check what pushes have just been compiled and then make a decision about the best code to generate.

The Virtual Stack

The code in Figure Two is superficially quite unlike traditional Forth code, in that it is not centered on the stack. But this is an illusion. The statement

```
-1 addto length
```

corresponds to code which:

1. loads register D0 with the value -1
2. pushes D0 to the parameter stack
3. pops the parameter stack to D0
4. adds the value in D0 to reg D5 (allocated to LENGTH).

The optimizer removes steps two and three, with the result that D0 is loaded with -1 and added directly to D5. In this example, the immediate value of -1 is only virtually pushed to the stack because of the relationship of the push to the code which follows it. The outcome is the conversion of four instructions (eight bytes, 34 cycles) to two instructions (four bytes, ten cycles).

The Two-edged Stack

A simple edge (push/pop) optimizer is able to carry out this code improvement. Its sole function is to remove unproductive PUSH/POP instruction pairs which result from adjacent macro expansions, and the resultant code is smaller and up to twice as fast. Examination of typical Forth code, however, suggests that we can do yet better. Consider the simple expression

```
3 4 +
```

The unoptimized code corresponding to this is in Figure Three-a. This is improved by a simple edge optimizer to the code in Figure Three-b. But it is clearly pointless to push the value four to the stack and then pop it again. Better code (Figure Three-c) is produced from the same macro primitives by a smarter optimizer which is able to look back at the previous edge and convert the push to a move to register D1, the secondary accumulator. This code pattern is very common in Forth words—like +, -, AND, OR, and the comparison words, each of which expects two edges on the stack.

Only the last two edges are remembered, so the optimizer works best with code which avoids a large number of consecutive pushes. It is therefore better to code

```
3 4 + 5 + 6 +
```

rather than

```
3 4 5 6 + + +
```

(it's easier to read, anyway).

Fetching and Storing

There are already too many fetch and store words in Forth and, if we were to add new words able to increment and decrement pointers, there would be three times as many. The cleanest approach seems to be to factor out the adjustment by using the prefix words INC and DEC to set a flag for enhanced versions of @ and !, which modify the macro code as it is expanded. But this is only part of the story.

By default, all fetch and store memory references are made with the main address register (A0). If, however, a memory pointer is stored in a named address register (such as SOURCE in Figure Four), it is a waste to have to transfer this pointer to A0 before using it. Rather, the address register should itself be used as the pointer. This is especially true if there is a pointer adjustment, otherwise it will be necessary to copy the altered address back to the register variable.

On the other hand, if a data register is used to hold the pointer, it cannot be used directly for memory reference and these register-to-register transfers are unavoidable. The result is code which is bulkier and less efficient (with two register-to-register moves). When local variables are used, INC and DEC can still be used, but the address must now be transferred from and to memory. The code is even slower (28 cycles) and bulkier (eight bytes), even though it is identical at the source level.

Only two of the six user-available registers are address registers (Figure One), and these are the first registers to be allocated. It turns out that this is just the best arrangement for the standard Forth memory words like CMOVE, CMOVE>, and FILL, where the address(es) come before other parameters.

The code produced by the compiler for definitions of these three words is almost as good as the best code an assembly programmer can write (Figure Five).

Implementation of the Improved Optimizer

The simple optimizer simply examines the last 16-bit value of the compile code to determine whether it is a PUSH. The new version uses specific bits in the FFA (flag field address) of each macro word as it is expanded. The present structure of the ForST header is given in Figure Six, and the utility word WHAT (Figure Seven) is provided for the programmer to examine the status of any of the system and compiled words. It is used as:

```
WHAT <name>
```

At the start of an expansion/optimization cycle, the compiler checks the present number of edges on the stack, and the addresses of the last two of them. If there happen to be two edges and the next word to be compiled expects two, the top edge is removed or modified to a register move (as before) with a saving of four or two bytes (and 24 or 18 clock cycles). The second edge is converted to a move to the secondary accumulator (register D1) with a further saving of 18 cycles.

It is simple enough to set up an array for 16-bit values for each of the registers, corresponding to PUSH, POP, ADDR, and ADDTO. The PFA of each of the temporary headers for the register variables has an offset into the array. A further offset is added by the prefix words PUSH, POP, etc.

Direct Register Access

The final development of the use of named register variables is to provide direct access to the hardware registers of the CPU. In ForST, this is presently limited to the Scratch and System groups (Figure One), and it is recommended that the System group be handled *very* cautiously.

Making use of the register words, macros, and the two-edged optimizer, it is possible to reconstruct quite efficient code primitives. While the present details are specific to the 68000, the ideas can be generalized to any processor which has access to two data registers (perhaps generically named D0 and D1) and two address registers (A0 and A1).

As implemented, there are simple macro categories in ForST:

1. D0 used only:
NOT, NEGATE, 0=

2. D0 and D1 used:
+, -, AND, OR, >, =, etc.
3. D0 and A0 used:
fetch and store words
4. A0 and A1 used:
CMOVE, CMOVE>
(One or more of these registers can be used as a temporary store in small definitions, but A1 is the only one which is relatively safe.)

Returning to our high-level definition of CMOVE (Figure Two-e) and the resultant object code (Figure Two-f), we see that significant code space (and some time) is taken up by the instructions which save and restore the three registers used in the definition. This is necessary in a high-level ForST definition, but we can achieve the same result more economically by direct access to registers (Figure Seven-a, Seven-b).

The scratch registers can be used to build up customized or duplicate versions of the Forth primitives (Figure Eight). It is very instructive to analyze the code from some of the different definitions of the same word. In the main, it compares well with assembly code and suggests that, except when specific Forth hardware is being used, the true primitives of Forth are not the well-known stack-juggling words, but a lower set of (potentially portable) register and memory operations.

Stand-alone Application Code

A practical outcome of this development is the ability to set up very efficient, completely free-standing applications which require no access to an underlying Forth system. Furthermore, if a modified compiler and macro primitives are loaded, code can be generated for any arbitrary target microprocessor.

Many of the utility words of Forth have been recoded and packaged into a number of included files analogous to the header files of C (Figure Nine).

Compilation definitions—including those which incorporate DELAYed compilation—are in a prelude file and, at the outermost level of the application, a file header is incorporated, together with code for reserving memory, setting up stacks, initializing registers and, at the end, exiting and freeing the allocated memory. All this is achieved in ForST without access to an assembler.

As a model for the approach, a small utility program (DUMP.TTP) was written. Its total file size is 1534 bytes, compared to a very similar program of over 10K bytes (presumably written in C) which is included in the Megamax Laser C system. It is used to provide a hex dump of disk files, which it does more than twice as fast as the Megamax program!

DUMP.TTP is an instructive model, as it includes text output, customized numeric output, keyboard input, and file input. All the necessary words were written using register variables and macro expansions.

The ability to generate efficient, totally independent code should be a useful development in Forth. During development, all the utilities of the Forth system are available, so development speed is not compromised and, at the end of it all, APPSAVE can be used to save the application to the disk.

Conclusion

High-level code which uses register variables is always smaller and much faster than that produced using the familiar stack operations. Operations with register variables implicitly use the Forth parameter stack, but the code optimizer minimizes the extent to which data are moved to and from the stack. There is still room for improvement of the optimizer, but code produced by it already compares well with that generated by the best compilers for other languages.

The availability of a compiler capable of using named register variables allows factoring of code—including that of familiar Forth code primitives—into a limited number of primitive CPU operations. Good quality code, including register initialization, can be achieved without access to an assembler.

John Redmond is an Associate Professor of Organic Chemistry at Sydney's Macquarie University. He is a "...sometimes-evenings-when-I-have-time programmer" who welcomes letters from FD readers: 23 Mirool Street, West Ryde, NSW 2114, Australia.

Scratch group

D0	Main accumulator
D1	Secondary accumulator
A0	Main pointer
A1	Secondary pointer

System group

D6	Loop index (I)
D7	Loop limit (I')
A4	Stack frame pointer (local variables)
A5	Code base pointer
A6	Parameter stack pointer (=SP)
A7	Return stack pointer (=RP)

User-available group (in order of allocation)
A3 A2 D5 D4 D3 D2

Figure One. Register allocation in ForST.

```
: cmove  { 3 regargs source dest length }
  length 0> if
     begin source c@ dest c!  ( transfer a byte)
       1 addto source 1 addto dest  ( increment pointers)
       -1 addto length  ( decrement length)
     length 0= until
  then ;
```

Figure Two-a. Example source-code definition.

```
: cmove   ( source,dest,length — )
  dup 0> if  >r
     begin  over c@ over c!
     1+ swap 1+ swap
     r> 1- >r
     r@ 0= until
  then  drop drop  r> drop ;
```

Figure Two-b. The code generated from Figure Two-a.

```
: cmove { 3 regargs source dest length }
  length 0> if
     begin source inc c@ dest inc c!
            -1 addto length
     length 0= until
  then ;
```

Figure Two-c. Using INC to direct postincrements.

```
: cmove { 3 regargs source dest length }
  for length  source inc c@ dest inc c!  next ;
```

Figure Two-d. Streamlined version with FOR as a local-variable prefix.

```
      movem.l a2/a3/d5,-(a7)   ;save the registers
      movem.l (a6)+,a2/a3/d5   ;get the arguments
      move.l  d5,d0     ;set the flags for length
      ble     .out    ;no good if zero or negative
 .lp: moveq.  l#0,d0        ;clear the accumulator
      move.b  (a3)+,d0         ;fetch the byte
      move.b  d0,(a2)+         ;and store it
      subq.l  #1,d5            ;count down
      bgt.s   .lp
 .out:movem.l (a7)+,a2/a3/d5;restore the registers
      rts
```

Figure Two-e. Object code generated by Figure Two-d.

```
: c>wmove  { 3 regargs source dest length }
  for length  source inc c@ dest inc w!  next ;
```

Figure Two-f. A "more eccentric" example to expand a byte array into a cell array.

```
moveq.l      #4,d0
push         d0
moveq.l      #3,d0
push         d0
pop          d0
pop          d1
add.l        d1,d0
push         d0
```

Figure Three-a. Unoptimized code to add two numbers.

```
moveq.l      #4,d0
push         d0
moveq.l      #3,d0
pop          d1
add.l        d1,d0
push         d0
```

Figure Three-b. Improved by a simple edge optimizer, but still wasteful.

```
moveq.l      #4,d0
move.l       d0,d1
moveq.l      #3,d0
add.l        d1,d0
push         d0
```

Figure Three-c. A smarter optimizer can use the previous edge intelligently.

Prefixes for local and register variables

(FROM)	Never used (default).
TO	Move from parameter stack to variable.
ADDTO	Add from parameter stack to variable.
ADDR	Address of local variable (*not* register) to parameter stack.
FOR	Use local or register variable as a down counter for a FOR ... NEXT loop.

Prefixes for fetch and store words

INC	Increment address pointer (by 1, 2, or 4, as appropriate) after a memory fetch or store.
DEC	Decrement address pointer (by 1, 2, or 4) before a memory fetch or store.

Figure Four. Summary of ForST prefixes.

```
: cmove   { 3 regargs source dest length }
  for length   source inc c@   dest inc c!   next ;

: cmove>  { 3 regargs source dest length }
  length addto source   length addto dest
  for length   source dec c@   dest dec c!   next ;

: fill   { 3 regargs pointer char length }
  for length   char pointer inc c!   next ;
```

Figure Five. Resulting definitions of some Forth memory words.

```
NFA:    length + $80 (or $C0), 'name', (+pad byte)
FFA:    edge bits*                  (bits 27-31)
        macro length (=bytes/2)     (bits 16-26)
        macro flag                  (bits 0-15 )
CFA:    offset address of code
PFA:    offset address of data or code


*Edge bits:     31:    one edge expected
                30:    two edges expected
                29:    one edge returned
                28:    two edges returned
                27:    Boolean result returned
```

Figure Six. Structure of the ForST header (revised).

```
: cmove  to d1 to a1 to a0
  for d1  a0 inc c@  a1 inc c!  next ;
```

Figure Seven-a. Direct register access brings more economy.

```
    pop                 d1
    pop                 a1
    pop                 a0
    move.l  d1,d1        ;set the flags for length
    ble       .out     ;no good if zero or negative
 .lp: moveq.l #0,d0           ;clear the accumulator
    move.b    (a0)+,d0        ;fetch the byte
    move.b    d0,(a1)+        ;and store it
    subq.l    #1,d1           ;count down
    bgt.s     .lp
 .out: rts
```

Figure Seven-b. Code generated by Figure Seven-a.

```
: dup  to d0 ( pop d0)  d0 d0 ( push d0 twice) ;
: dup  sp @  ( push what is on top of stack) ;
: ?dup sp @ to a1 a1 if ( non-zero) a1 ( push sec-
ond copy) then ;
: >r  rp dec ! ;
: r>  rp inc @ ;

: count  to a0  a0 inc c@ to d0 ( hold in d0)
  a0 ( push) d0 ( push) ;

: drop  to d0 ;
: drop  4 addto sp ;

: over  sp 4+ @ ;
: over  to d0 to d1  d1 d0 d1 ;
: over  to d1  sp @ to d0  d1 d0 ;

: tuck  to d0 to d1  d0 d1 d0 ;

: nip  to d0 to d1  d0 ;
: nip  sp ! ;

: rot  to a1 to a0 to d0  a0 a1 d0 ;
: -rot  to a1 to a0 to d0  a1 d0 a0 ;

: aligned  sp @ 1 and + ;
: aligned  to a1  a1 1 and addto a1  a1 ;

: fill  to d0 ( char) to d1 ( length) to a0 ( ad-
dress)
  for d1  d0 a0 inc c!  next ;
: cmove> to d1 ( length) to a1 ( dest) to a0 ( source)
 d1 addto a0 d1 addto a1 ( point to ends of strings)
  for d1  a0 dec c@ a1 dec c!  next ;
```

Figure Eight. Using scratch registers to customize or duplicate Forth primitives.

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

```
\ DUMPAPP.S: application demo program

  decimal macros

  cd a:\forth       ( set current directory)

  load appskel.s    ( prelude for applications)
  load filehead.s   ( file header and entry/exit
                    code)
  load apputils.s   ( general utilities)
  load conio.s
   head type is >type ( type defined in conio.s)
   head (") is >(")  ( also in conio.s)
  load intout.s     ( integer output)
   head . is >.
  load appfilin.s   ( disk file input)
  load dump.s       ( the actual application)
```

Figure Nine. Including Forth utilities via header-like files.

Advertisers Index

(Continued from page 23.)

```
\
\   7 6 5 4 3 2 1 0 -Data Terminal Ready (DTR)
\   | | | | | | +----Request to Send (RTS)
\   | | | | | +------Out 1
\   | | | | +--------Out 2
\   | | | +----------Loop
\   | | +------------=0
\   | +--------------=0
\   +----------------=0
\
\   Line Status Register   LSR  com.base 5 +
\
\   7 6 5 4 3 2 1 0 -Data Ready (DR)
\   | | | | | | +----Overrun Error (OR)
\   | | | | | +------Parity Error (PE)
\   | | | | +--------Framing Error (FE)
\   | | | +----------Break Interrupt (BI)
\   | | +------------Transmitter Holding Register Empty (THRE)
\   | +--------------TX Shift Register Empty (TSRE)
\   +----------------=0
\
\   Modem Status Register   MSR  com.base 6 +
\
\   7 6 5 4 3 2 1 0 -Delta Clear to Send (DCTS)
\   | | | | | | +----Delta Data Set Ready (DDSR)
\   | | | | | +------Trailing Edge Ring Indicator (TERI)
\   | | | | +--------Delta RX Line Signal Detect (DRLSD)
\   | | | +----------Clear To Send (CTS)
\   | | +------------Data Set Ready (DSD)
\   | +--------------Ring Indicator (RI)
\   +----------------Receive Line Signal Detect (RLSD)
```

GENIE FOR BEGINNERS

FRANK C. SERGEANT - SAN MARCOS, TEXAS

If you haven't broken into bulletin boarding yet, now is the time! It is easier than you think and it just got cheaper. I'll show you how I do it, in hopes of encouraging you to try too.

Month after month, you've been seeing GEnie and the other Forth bulletin boards listed in *Forth Dimensions*. If you are like me, you've been planning to try it sometime, but it is one project too many. You've put it on the back burner. Projects that confuse us get put off more easily than ones we understand fully. I was that way about GEnie. It was a strange new world and I didn't know how to start. It turned out to be pretty simple.

GEnie is the on-line "home" of the Forth Interest Group, where we meet to exchange ideas, ask questions, get help, express ourselves, and keep up with the latest Forth news. GEnie is also the host of many other special interest groups, on-line shopping, news, stock quotes, games, and electronic mail (E-mail). For me, the Forth RoundTable and E-mail are the most important. One just got cheaper and the other almost free!

First things first: you need a computer. Since you are a Forth enthusiast, you probably already have one. Next, you need a modem to connect your computer, via the telephone lines, to GEnie. These are getting pretty cheap. They come in several speeds, the faster the better. The faster ones can also run at the slower speeds if necessary, so you don't give up anything but money to go with a faster modem. There are two types: internal and external. If you have a PC/XT clone you are especially lucky. Because of its large market and competition, you can get an internal 2400 bps (bits per second) modem for under $80.00. (For example, check with Hard Drives at 1-800-766-DISK. Ask for Eric and tell him I sent you.) If possible, get one with at least a speed of 1200 bps, preferably 2400 bps. After that, they get considerably more expensive. The external modems will work with any computer that has a serial port. If it's all you can afford, at least get a 300 bps external modem—it should be awfully cheap these days.

Third, you need telecommunication software, often called a comm program. Basically, you want a program that will copy whatever you type to the modem and from the modem to your screen, all the while saving a copy to disk so you can read it later. The modem connects your computer, through the phone lines, to GEnie. Perhaps we should do this in Forth, but until then it sure is easy to get one of the many shareware programs such as PROCOMM or QMODEM (perhaps from Public Brands at 1-800-426-DISK or from PsL at 1-800-2424-PSL or from any high school student with a computer). You save a copy to disk so you can read it at your leisure, when you are not tying up the phone (and spending money). For my communication software I use FlashLink. It came with the modem from Hard Drives and I've been very happy with it.

They're making us a deal we can't refuse.

Fourth, you need to sign up with GEnie. This is pretty easy. Phone GEnie Client Services at 1-800-638-9636 to talk to a friendly human and get your local GEnie phone number. Yup! From most places you can reach GEnie with a local phone call. Bring up your comm program and set it for half duplex (i.e., local echo). If you have an MS-DOS machine, set it for seven bits, even parity. Otherwise, use eight bits, no parity. Set your terminal emulation to use TVI (Televideo) if possible (I use VT102 for FlashLink). If you have any trouble doing this, don't despair, call Client Services. They'll help you get the modem and comm program set up right (they seem to want your business). Now have your modem dial the local number or 1-800-638-8369 to sign up. When the modem says CONNECT, type 3 H's (HHH) followed by a carriage return; then GEnie will reply with something like U# (for "user number"). For first-time signers up, type SIGNUP and press return. GEnie will prompt you through the sign-up process. You'll need your checkbook or credit card handy.

Go through the same procedure to log on in the future, except dial your local access telephone number, and type your actual user number and password (as given to you during the initial log-on process).

This is a great time to start. GEnie had been charging five dollars per hour for 300 bps, six dollars per hour for 1200 bps, and ten dollars per hour for 2400 bps. As of October 1, they changed to the new Star Services plan. At first I said, "Uh-oh, they're going to *help* me by charging me more money." I'm suspicious that way, but for once it was unfounded. As I see it, they're making us a deal we can't refuse. For a flat rate of $4.95 a month, we get unlimited access to about half of GEnie. This includes the E-mail section, encyclopedia, shopping mall, and the "leisure and professional" RoundTables. Unfortunately, the Forth RoundTable is not in that half. Nevertheless, they have reduced their charges here also (get a 2400 bps modem if you don't have one yet). In the computer RoundTables, the charge is now only six dollars per hour—even at 2400 bps. So the change is still a good deal. For only $4.95 a month you and all your friends around the country, your parents, your children, and your business associates should sign up, if only for the E-mail. Then, except for sending photographs, you can practically eliminate your dealings with the U.S. Postal Service (and even fax machines). With a local phone call and no charge per hour, you could even live with a 300 bps modem for the E-mail services. And there is no longer a sign-up fee! These rates apply to non-prime time, which is between 6 p.m. and 8 a.m. local time and 24 hours on weekends and holidays. Stay off during the day—it costs a hefty $18 per hour then. One of the nice things about the new system for beginners is that, for no hourly charge at all, you can *practice* in one of the free RoundTables. You can learn your way around the bulletin board without pressure from the sound of the cash register ticking! Then apply your new skills when you log into the Forth RoundTable.

Some people, such as myself, live so far from civilization that there is a communication surcharge of two dollars per hour for that local call. Still, that beats long-distance rates. I was surprised to find GEnie had a local number for me at all.

Barriers to Overcome

The hardest part is starting. Once you sign up and learn enough about your comm program to actually log on, all you have to do is follow the menus. Call the Client Services number for help if you get stuck. Once you learn to log on, you'll want to start reading the messages. The next step up is to reply to a message. Don't put this off. The first one is the hardest. Then comes E-mail. After that comes uploading and downloading files. Take it one step at a time.

Set Your Break Key

If you didn't do this as part of the sign-up procedure, type TOP at any prompt to get to the top menu. Or work your way back up to the top, from wherever you are, by repeatedly selecting P (for "previous") from the current menu. Then get to the break key setup by selecting, consecutively, #3 Billing/Setup, #3 Settings, #1 Terminal settings, #2 Terminal settings, and finally #2 Break char. Then type the ASCII value in decimal for the key you want to use for your GEnie break key. I had been using 03 (for Ctrl-C) with my previous comm program, but FlashLink wouldn't pass that key on to the modem, so I changed mine to 126 (for the ~ key). Then select #9 Save and return.

Reading Messages

This is pretty easy. Get into the Forth RoundTable by typing FORTH or by typing m710;1 at any "Enter #, <P>revious, or <H>elp?" prompt. You'll start off in category 1. You have a hard decision to make: Should you read all messages that have been posted from the beginning of time, or should you start fresh? I started from the very beginning. It took a long time to read them all. I'm glad I did, but it might not be the best way for you. If you want to read them all, type BRO ALL NOR (for "browse all no-reply"). This will display all of the new messages (i.e., the ones you haven't read yet) in all of the topics in all of the categories of the Forth RoundTable. All the messages will be new to you when you start. Be prepared, there are a *lot* of them! When you get tired, hit the break key and quit for the day. (To log off, just type BYE at the prompt.) It will take you days to get them all.

For many people, a more sensible approach is to ignore all the previous messages. In that case, type IGN ALL. Next time you log on, type BRO ALL NOR (as described above) and you'll get all of the messages posted after you ignored them all. Suppose you do an IGN ALL and don't want to wait but want to read a few messages right now? You're already in CAT 1 (you can tell by the prompt), so type REA ALL DAT>901115 NOR to read all of the topics in the current category that have a date greater than November 15, 1990, without stopping for your reply to each message. Naturally, you can use whatever date suits you. Then, pick another category, such as number 2, by typing SET 2, and do it again with REA ALL DAT>901115 NOR (or use another date) and so on until you've read all the categories you're interested in. Thereafter, use the BRO ALL NOR method. GEnie keeps track of the messages you've already read (or ignored). Note that BROwse reads all new messages in all categories, while the REAd command is restricted to the current category.

After I log off, I use QEDIT to browse through the file to read the messages. [Most text editors can read the ASCII, or text-only, files captured while on line.] I copy the ones of special interest to another file. I have a set of files named *.SUM (for "summary") into which I collect the most memorable messages. This is very easy to do with QEDIT (a shareware program also available from the above-mentioned distributors) as it lets me edit many files at one time.

Posting Messages

It may be hard to get up the nerve to post your first message, but after that it gets easy. Here's how I do it: I browse through my capture file until I find one I want to reply to. I compose my reply with QEDIT (use your favorite editor) and save it to disk [as an unformatted, ASCII file with carriage returns at the end of each line]. I save my reply to a disk file whose name indicates the category number and topic number it pertains to. For example, if I am replying to a message about Pygmy Forth, which is in Category 1, Topic 45, I might name the file C1T45RB1. Next time I log onto GEnie I upload the reply. Then I rename the file to C1T45RB1.SNT (for "sent") so I'll know I've posted it.

To upload the reply, I first get into the proper category with the SET command (e.g., SET 1). Then I tell GEnie what topic the message should go to by saying REP45 (for "reply to topic 45"). GEnie will then offer to let me enter the first line of the reply. Instead of typing the message, I type *U to say I want to upload the reply rather than typing it from the keyboard. When GEnie says it is ready for the upload, I press the PgUp key (this is how FlashLink works—check the instructions for your comm program), select ASCII, and type the name of the file to upload (in this example it would be C1T45RB1). Then GEnie and the comm program transfer the file. When it is complete, I press the break key (a "~" in my case). When GEnie comes back with a line number, I can type *L to list it on my screen (I usually don't bother) and then *S to send it (if you don't do this, it won't get "sent").

That's all there is to it. You'll get the hang of all this pretty quick if you'll just dive in and try it. Please don't be bashful about posting messages. The other users of the RoundTable are always willing to help correct your errors in thinking! Often, they are surprisingly polite, no matter how wrong you are.

Downloading, Uploading, E-mail

You might also want to download

(Continued on page 41.)

BEST OF GENIE

GARY SMITH - LITTLE ROCK, ARKANSAS

News from the GEnie Forth RoundTable—For those who have wondered about the sudden pause in ForthNet ports from the xCFB's (the PC-board branch of ForthNet), here finally is an explanation. Work requirements and years of dedication to the original xCFB, the East Coast Forth Board, finally took its toll on ECFB SysOp Jerry Shifrin; he elected to call it quits. Despite personal need to the contrary, Jerry stuck it out long enough for us to locate a new central node for the xCFB's that would have a SysOp willing to deal with the mail porting necessary to maintain the ForthNet bridge.

We have located such a BBS and such a SysOp in Jim Wenzel of the nationally recognized Grapevine, located in Little Rock, Arkansas. While Grapevine is *not* accessible via PC-Pursuit, it is accessible via StarLink on node 9858. The Forth conference on Grapevine is 58.

Jim's BBS is a PC board and maintains storage for two gigabytes. Jim was already a major node in the RIME network, so we are fortunate to have him and Grapevine as members of ForthNet. For those interested in accessing Grapevine directly, registration is via data phone number 501-753-8121. Once registered, use 501-753-6859. Any baud rate up to 19200 is supported on either line. Line 753-6859 rolls over if busy and if the other line is open.

We owe a great debt of gratitude to Jerry Shifrin for his tireless dedication all this time. I can absolutely guarantee, if not for Jerry there would be no ForthNet. Jerry started ECFB when he was, indeed, a single light in the darkness; and he was a tremendous help and influence when we began the long process of building what is now ForthNet.

Good luck in whatever your future endeavors may be, Jerry. We will all miss you. I can promise I will.

Welcome to the club Jim Wenzel. We are lucky someone as dedicated as you was willing to take up the slack.

*   *   *

Two expressions pop to mind as I realize how long it has been since I recapped the GEnie Forth RoundTable real-time guest conferences. The first is, "Time sure flies when you are having fun." The second has something to do with posteriors, swamps, and alligators. These trips down memory lane are without doubt one of the more popular features of this column, and certainly they are among my favorite to date.

I have said this before, but it bears repeating. We have really been graced by an array of guests with divergent views, but without exception all have been interesting and a delight to chat with. I would encourage you to participate in at least one of these conferences. The guests generally appear one per month, on the third Thursday except during the fourth quarter (October through December, when they are moved to the second Thursday to avoid holiday conflicts). If you are interested in knowing who might be scheduled to appear next, this is posted on the GEnie Forth RoundTable Bulletin Board in Category 1, Topic 6 with the current invitee pre-announced in the "door banner" that greets users on entrance to the RoundTable each day. Enough of that. We have a lot of catching up to do.

We have the organization and it has room for new directions.

This issue we will focus on visits with Robert Smith, "Floored Division and Floating Point"; Phil Koopman, "Stack Machines"; Charles Curley, "A Minimalist View of Forth"; and John D. Hall, "The Business of FIG." As has become a standard format for these guest sessions, I will recall only the guests' opening remarks. These can be used quite accurately as a pseudo-abstract of the conference. If you want to follow the discussion more closely, the complete transcripts are available for capture in Library 1 of the GEnie Forth RoundTable Software Libraries.

Phil Koopman (September 1989)
Senior Scientist, Harris Semiconductor

Some of the things I have found out about stack machines go against widely held (at least, outside the Forth community) ideas. For example, stack machines: don't need stacks bigger than 16 to 32 elements, need not have a significant context switching time, and can cycle their clocks every bit as fast as (or perhaps faster than) RISC processors. One thing I run across continually is that folks confuse the requirements for real-time embedded control with those of workstation environments. One of my professional goals is to understand more about Forth-derived stack computers in order to help them gain acceptance in applications for which they are well suited. Stack machines seem to be superb at real-time embedded control (although I still want to do more research to quantify this notion). But, what about other application areas? If stack machines are the answer, what are the questions?

Robert L. Smith (October 1989)
Research Specialist, Lockheed Palo Alto

Thank you. For floored division, it helps to focus on the modulus or remainder rather than the quotient. Most users use only positive arguments, so floored or non-floored give the same results. For almost all cases that I know of, if you have at least a negative numerator, you probably should use floored division.

For floating point:

(1) should Forth have it at all? And

(2) if so, should it be in the standard? And

(3) IEEE floating point?

Charles Curley (November 1989)
Neologist

Grief, after that lead-in. I guess I see Forth as a tool, and I like my tools to be simple and easy to understand. A lathe is (conceptually) simple, and it does almost anything one needs to do. So are wrenches, hammers, etc. I don't think there is such a thing as a single tool to do everything, so I like lots of simple, semi-custom tools. Hence minimal Forth.

That's all for a start.

<[Gary] GARY-S> Charles, your original non-standard committee is something of a standard rejoiner. Do you still feel anti-standard?

Well, since no-one ever joined it, it can't be a <re->joiner <grin> but, yes, I do still think that standards get in the way more than they help. Note that standards are not the same as models, which can be very useful.

I think Chuck said it best: standards are wonderful, everyone should have one. I think that coding standards would be more useful than a language standard, but try getting any two Forth programmers to meet even minimal standards in coding... I think standards are too vague to be of any real use. They are too open to interpretation and fudges. If they are too tight, the resultant Forth is useless for its natural applications, high speed and compact code stuff. If they're too loose, then you have no portability.

John D. Hall (December 1989)
Programmer, Lockheed Palo Alto

The Forth Interest Group was organized in 1978, and one of the first things done was the formation of the Forth Implementation Team led by Bill Ragsdale to build fig-FORTH and put it in the public domain. Because fig-FORTH was implemented on many microprocessors, based on a single model and released with complete source listings, it became the *de facto* standard of Forth. The listing were made available for $15 and were extremely popular. To encourage extensions and modifications to Forth, the same people started *Forth Dimensions*, FORML, and the Forth Standards Team. Many of these same people wrote or encouraged others to write articles for *Byte*, and the famous *Byte* Forth issue was published in October of 1980. Money was coming in faster than new uses for it could be found and sales taxes were collected, but no formal organization existed to pay the sales tax and to avoid income tax. At that time, FIG was only a loose confederation of interested Forth users around the world, so a California non-profit corporation, Forth Interest Group, Inc., was formed to centralize the contacts, establish a distribution point for Forth information, and establish the formal function of publishing *Forth Dimensions* and the FORML proceedings, and to distribute these and the fig-FORTH listings. No great thought was given to the other reasons for having a corporation. It was primarily to centralize the loose confederation of people and ideas and to distribute contributed literature. FIG was guided by a Board of Directors consisting of Bill Ragsdale, Kim Harris, John James, Dave Kilbridge, and Dave Boulton.

The corporation wasn't the answer to the problems of organization, it took people to make ideas happen. Organization ideas were and still are plentiful, people to implement the ideas were not. From the beginning, business meetings were held one evening a month on the Tuesday before the fourth Saturday of each month. Before the corporation, FIG and the Silicon Valley chapter were the same, and the planning for the Saturday meeting was finalized on the previous Tuesday. As the corporation was formed, the corporation business meeting began to take over the topics of the Tuesday meeting until finally the Silicon Valley chapter planning was split off to another night. Since actions took more time than was available at the business meetings, committees were formed to plan and guide activities as existing or new functions were needed. FORML and *Forth Dimensions* needed guidance. FORML split into a conference and a convention. People around the world were forming group meetings and wanted to be kept informed, so a chapter committee was formed. Literature needed to be published, and new books and publications needed to be reviewed; a publication committee was formed. Direct communication with the Forth community was needed, so a connection to GEnie was established and a committee of sysops was formed. All of the literature needed to be distributed and orders needed to be filled, so an outside organization was hired. Money was being received and distributed on a daily basis and this was more than the volunteer treasurer could handle. From the beginning, all the functions of FIG, Inc. were handled by a group of volunteers with the help of a few paid people such as the editor of *Forth Dimensions* and Mountain View Press for distribution. When we saw our way clear enough, a management organization was hired to try to tie all the office functions together to establish a central office for mail distribution and phone, under the direct control of FIG, Inc.

We are now at a point now where FIG, Inc. includes

1) Editing, publication, and distribution of *Forth Dimensions*
2) Producing and convening the annual FORML conference
3) Publication and distribution of the FORML Proceedings
4) Producing and convening the annual Forth convention
5) Support and sponsorship of the GEnie RoundTable
6) Support and organization of approximately 50 chapters
7) Distribution of over 90 publications
8) Production and distribution of a growing disk library
9) Membership enrollment of over 2000 members

FIG, Inc. is governed by a seven-member Board of Directors: Dennis Ruffer, John Hall, Terri Sutton, Mike Elola, Robert Smith, Jack Brown, Wil Baden. Business is conducted by the Business Group, including the above and: C.H. Ting, Jan Shepherd, Bob Barr, Marlin Ouverson, Bill Ragsdale, Robert Reiling, Tom Zimmer, and others who are interested in the business of FIG.

The growth of FIG was probably not as chaotic as I am remembering it, but the organizing was driven by needs that had to be filled rather than by a planned structure, and the organizing was done by those of us who are technically competent in Forth but not necessarily in organizations.

(Continued on page 39.)

REFERENCE SECTION

Forth Interest Group
The Forth Interest Group serves both expert and novice members with its network of chapters, *Forth Dimensions*, and conferences that regularly attract participants from around the world. For membership information, or to reserve advertising space, contact the administrative offices:

Forth Interest Group
P.O. Box 8231
San Jose, California 95155
408-277-0668

Board of Directors
Robert Reiling, President *(ret. director)*
Dennis Ruffer, Vice-President
John D. Hall, Treasurer
Wil Baden
Jack Brown
Mike Elola
Robert L. Smith

Founding Directors
William Ragsdale
Kim Harris
Dave Boulton
Dave Kilbridge
John James

In Recognition
Recognition is offered annually to a person who has made an outstanding contribution in support of Forth and the Forth Interest Group. The individual is nominated and selected by previous recipients of the "FIGGY." Each receives an engraved award, and is named on a plaque in the administrative offices.

1979 William Ragsdale
1980 Kim Harris
1981 Dave Kilbridge
1982 Roy Martens
1983 John D. Hall
1984 Robert Reiling
1985 Thea Martin
1986 C.H. Ting
1987 Marlin Ouverson
1988 Dennis Ruffer
1989 Jan Shepherd

ANS Forth
The following members of the ANS X3J14 Forth Standard Committee are available to personally carry your proposals and concerns to the committee. Please feel free to call or write to them directly:

Gary Betts
Unisyn
301 Main, penthouse #2
Longmont, CO 80501
303-924-9193

Mike Nemeth
CSC
10025 Locust St.
Glenndale, MD 20769
301-286-8313

Andrew Kobziar
NCR Medical Systems Group
950 Danby Rd.
Ithaca, NY 14850
607-273-5310

Elizabeth D. Rather
FORTH, Inc.
111 N. Sepulveda Blvd., suite 300
Manhattan Beach, CA 90266
213-372-8493

Charles Keane
Performance Packages, Inc.
515 Fourth Avenue
Watervleit, NY 12189-3703
518-274-4774

George Shaw
Shaw Laboratories
P.O. Box 3471
Hayward, CA 94540-3471
415-276-5953

David C. Petty
Digitel
125 Cambridge Park Dr.
Cambridge, MA 02140-2311

Forth Instruction
Los Angeles—Introductory and intermediate three-day intensive courses in Forth programming are offered monthly by Laboratory Microsystems. These hands-on courses are designed for engineers and programmers who need to become proficient in Forth in the least amount of time. Telephone 213-306-7412.

On-Line Resources
To communicate with these systems, set your modem and communication software to 300/1200/2400 baud with eight bits, no parity, and one stop bit, unless noted otherwise. GEnie requires local echo.

GEnie
For information, call 800-638-9636
- Forth RoundTable
  (ForthNet link)*
  Call GEnie local node, then type M710 or FORTH
  SysOps: Dennis Ruffer (D.RUFFER), Scott Squires (S.W.SQUIRES), Leonard Morgenstern (NMORGENSTERN), Gary Smith (GARY-S)
- MACH2 RoundTable
  Type M450 or MACH2
  Palo Alto Shipping Company
  SysOp: Waymen Askey (D.MILEY)

BIX (ByteNet)
For information, call 800-227-2983
- Forth Conference
  Access BIX via TymeNet, then type j

forth
Type FORTH at the : prompt
SysOp: Phil Wasson (PWASSON)
- LMI Conference
Type LMI at the : prompt
Laboratory MicroSystems products
Host: Ray Duncan (RDUNCAN)

CompuServe
For information, call 800-848-8990
- Creative Solutions Conference
Type !Go FORTH
SysOps: Don Colburn, Zach Zachariah, Ward McFarland, Jon Bryan, Greg Guerin, John Baxter, John Jeppson
- Computer Language Magazine Conference
Type !Go CLM
SysOps: Jim Kyle, Jeff Brenton, Chip Rabinowitz, Regina Starr Ridley

Unix BBS's with forth.conf (ForthNet links and reachable via StarLink node 9533 on TymNet and PC-Pursuit node casfa on TeleNet.)*
- WELL Forth conference
Access WELL via CompuserveNet or 415-332-6106
Fairwitness: Jack Woehr (jax)
- Wetware Forth conference
415-753-5265
Fairwitness: Gary Smith (gars)

PC Board BBS's devoted to Forth (ForthNet links)*
- British Columbia Forth Board
604-434-5886
SysOp: Jack Brown
- Grapevine
501-753-8121 to register
501-753-6389
StarLink node 9858
SysOp: Jim Wenzel
- Real-Time Control Forth Board
303-278-0364
StarLink node 2584 on TymNet
PC-Pursuit node coden on TeleNet
SysOp: Jack Woehr

Other Forth-specific BBS's
- Laboratory Microsystems, Inc.
213-306-3530
StarLink node 9184 on TymNet
PC-Pursuit node calan on TeleNet
SysOp: Ray Duncan
- Knowledge-Based Systems
Supports Fifth
409-696-7055
- Druma Forth Board
512-323-2402
StarLink node 1306 on TymNet
SysOps: S. Suresh, James Martin, Anne Moore
- Harris Semiconductor Board
407-729-4949
StarLink node 9902 on TymNet (toll from Post. St. Lucie)

Non-Forth-specific BBS's with extensive Forth Libraries
- College Corner (PC Board)
206-643-0804
300-2400 baud
SysOp: Jerry Houston
- Psymatic BBS
Sunnyvale, California
408-992-0372
300 – 2400 baud
This is a programmer's board with a large Forth area.

International Forth BBS's
- Melbourne FIG Chapter
(03) 809-1787 in Australia
61-3-809-1787 international
SysOp: Lance Collins
- Forth BBS JEDI
Paris, France
33 36 43 15 15
7 data bits, 1 stop, even parity
- Max BBS *(ForthNet link*)*
United Kingdom
0905 754157
SysOp: Jon Brooks
- Sky Port *(ForthNet link*)*
United Kingdom
44-1-294-1006
SysOp: Andy Brimson
- SweFIG
Per Alm Sweden
46-8-71-35751
- NEXUS Servicios de Informacion, S. L.
Travesera de Dalt, 104-106, Entlo. 4–5
08024 Barcelona, Spain
+ 34 3 2103355 (voice)
+ 34 3 2147262 (modem)
SysOps: Jesus Consuegra, Juanma Barranquero
barran@nexus.nsi.es (preferred)
barran@nsi.es
barran (on BIX)

This list was accurate as of October 1990. If you know another on-line Forth resource, please let me know so it can be included in this list. I can be reached in the following ways:

Gary Smith
P. O. Drawer 7680
Little Rock, Arkansas 72217
Telephone: 501-227-7817
GEnie (co-SysOp, Forth RT and Unix RT): GARY-S
Usenet domain.: uunet!wugate!
wuarchive!texbell!
ark!lrark!gars

**ForthNet is a virtual Forth network that links designated message bases in an attempt to provide greater information distribution to the Forth users served. It is provided courtesy of the SysOps of its various links.*

(Continued from page 37.)

Was FIG the answer for Forth?

Well, yes, if the question was how do we establish a central focal point for Forth. What we had done is created a "passive" organization, given it the ability to reach out to the Forth community and made it financially self-supporting as long as there were enough people who were interested in Forth and willing to support FIG.

What we had not done was create an active environment for encouraging those who were interested in Forth, instructing those who were new to Forth, supporting the creativity of people who were advancing Forth or even establishing the solid technical foundation and background for the growth of Forth use.

There are probably other things the FIG is not and should be.

We have the organization and it has the room for new directions. How do we now make it the organizational answer for Forth?

VOLUME XI INDEX

MIKE ELOLA - SAN JOSE, CALIFORNIA

(Continued on next page.)

(Continued from page 35.)

shareware, public-domain software, or transcripts of the guest conferences. A lot of files are available, including public-domain and shareware Forth systems for most processors. To download files, I usually type m711;4 at any "Enter #, <P>revious, or <H>elp?" prompt. This moves to the files section and selects the browse option. I start at the most recent file by pressing Enter. GEnie shows me a file description for each file and gives me the chance to download it or skip it. Skip it by pressing Enter. Download it by pressing D. If downloading, it will ask you what method to use. I pick YMODEM (with FlashLink) but use whatever you have available with your comm program, preferably ZMODEM. When GEnie says it's ready, I press PgDn to get FlashLink to start the download process. I again pick YMODEM (the first time was to tell GEnie, this time it is to tell FlashLink) and away they go. When the transfer is complete, I press the break key and carry on, skipping or downloading file after file. When I get to files that look familiar (i.e., older ones that I have browsed through previously) I quit.

To read or send E-mail, type m200 and follow the menu. To upload a file, type m711 and follow the menu.

Aladdin

GEnie has a special comm program available for MS-DOS machines, named Aladdin, that helps to automate the process of dealing with GEnie. I tried it briefly and did not like it. I do not use it. I believe many people like it; I may have quit too soon. Instead, I use FlashLink to capture my entire on-line session to disk, and later browse and create replies with QEDIT, and organize the most interesting saved messages into summary files (DSP.SUM, MATH.SUM, IMPLEM.SUM, PYGMY.SUM, etc.)

Summary

Bulletin boarding gives you access to a *lot* of expertise. You get to follow current issues about Forth with a rapid turn-around time. If you run into a problem, you usually get a quick reply and a lot of help. One fellow wanted a Forth assembler for the 68HC11, so I wrote him one and posted it to the bulletin board. (It makes a good joke once, but I probably won't do it for everyone who wants a new assembler!)

If you are a beginner to Forth (or not a beginner) you can learn a lot by participating in the bulletin board. I can't recommend it highly enough. Sure, you'll be a little uncomfortable until you get the hang of it, but it's worth it. I hope you'll give it a try *tonight*! You can reach me by sending GEnie E-mail to F.SERGEANT. I hope to hear from you soon.

Frank Sergeant is a hardware and software consultant specializing in business and real-time systems. He is the author and implementor of Pygmy Forth.

(Index, continued from previous page.)

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 Anna Brereton 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

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

  North Bay Chapter
  3rd Sat.
  12 noon tutorial, 1 p.m. Forth
  2055 Center St., Berkeley
  Leonard Morgenstern
  (415) 376-5241

  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.
  Applied Bio Systems
  Foster City
  (415) 535-1294

  Stockton Chapter
  Doug Dillon (209) 931-2448

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

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

  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.
  (708) 713-5365

  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
  3rd Wed., 6:30 p.m.
  JHU/APL, Bldg. 1
  Parsons Auditorium
  Mike Nemeth (301) 262-8140 (eves.)

- **MASSACHUSETTS**
  Boston FIG
  3rd Wed., 7 p.m.
  Bull HN
  300 Concord Rd., 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**
  Long Island Chapter
  3rd Thurs., 7:30 p.m.
  Brookhaven National Laboratory
  AGS dept., bldg. 911, lab rm. A-202
  Irving Montanez
  (516) 282-2540

Rochester Chapter
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
03/889-2600
BBS: 61 3 809 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
Lariksdreef 20
2120 Schoten
03/658-6343

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

- **CANADA**
FORTH-BC
1st Thurs., 7:30 p.m.
BCIT, 3700 Willingdon Ave.
BBY, Rm. 1A-324
Jack W. Brown
(604) 596-9764 or
(604) 436-0443
BCFB BBS (604) 434-5886

Northern Alberta Chapter
4th Thurs., 7–9:30 p.m.
N. Alta. Inst. of Tech.
Tony Van Muyden
(403) 486-6666 (days)
(403) 962-2203 (eves.)

Southern Ontario Chapter
Quarterly: 1st Sat. of Mar., June, and Dec. 2nd Sat. of Sept. 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

- **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)

- **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**
Tokyo Chapter
3rd Sat. afternoon
Hamacho-Kaikan, Chuoku
Toshio Inoue
(81) 3-812-2111 ext. 7073

- **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

SPECIAL GROUPS

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

1991 ROCHESTER FORTH CONFERENCE ON AUTOMATED INSTRUMENTS

June, 1991

University of Rochester
Rochester, New York

Call for Papers

There is a call for papers on the use of Forth technology in Automated Instruments. 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 1, 1990 and final papers by June 1, 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)

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

Second Class
Postage Paid at
San Jose, CA