$10 Volume XIV, Number 4 November 1992 December

FORTH DIMENSIONS

Hardware Interrupt Handler

Tutorial: Character Graphics

Styling Forth to Preserve C's Expressiveness

Principles of Metacompilation (II)

Contents

Features

Departments

Editorial

We had just decided to give readers a respite from ANS Forth's labor pains when we received a letter by Chuck Eaker. In it, he challenges Forth experts who are up to their necks in the standardization debates to turn to another worthy, rewarding, and perhaps more difficult task. Coincidentally, columnist Mike Elola passed this month's "Fast Forthward" essay space to fellow Board member Jack Woehr, instead developing an article closely related to Eaker's letter.

This issue's other contents range from a tutorial introduction to Forth to an 8051 assembler, an interrupt handler, and metacompilation. But if you're too experienced to need a tutorial, and too jaded to learn from others' work with metacompilation, start with Eaker's letter and Elola's article; if you take them seriously, we think you'll have your hands full.

The next issue will publish winners of our "Forth on a Grand Scale" contest. The object was to describe Forth projects of an unusually large or complex nature, and the top authors succeeded handily. We look forward to sharing their work with you.

We hope you will give serious thought to writing for *Forth Dimensions*. As a publication that is both by and for the Forth community, it rests on each of us to create an informative and useful publication. Tell us what you are doing with Forth, share your discoveries and obstacles, teach the rest of us something we should know.

* * *

As you may know too well, a peril of the self-employed worker is the persistent lack of "down time." The telephone rings at international hours; there is seldom anyone to delegate tasks to; and every time you pass the office door, a twinge of conscience strikes—there's always some task clamoring for your attention. Paid vacations and benefits? Forget about them.

Sometimes the only way to really take off work is to take off literally, and even that doesn't always work, not entirely. I recently left office and work (except for calls to the printer) for the first time since I don't remember when. Taking to the road, I ended up at a small encampment on a mountainous, native American reservation near the Canadian border. Nothing better counterracts a long-term, low-level overdose of technology than big sky, fresh air, spring water, general hilarity, and ceremonial observances of the unity of diverse people, their spiritual traditions, and the nurturing earth.

A sign was posted to help new arrivals find their way over the winding, unmarked roads. A family of Romanian expatriates chanced upon the gathering and found itself welcomed into a culture they had studied in books but never experienced. I overheard the father tell someone he is an engineer, and the technophile in me—not entirely exorcised—introduced itself to him. What a strange surprise, there among the jagged peaks and native culture, to meet a man who, when he came to the United States, was required to learn Forth for his first job.

We discussed how hardware has changed: the entire Romanian financial system once was maintained on a 256K computer (no documentation) with four washing-machine-sized hard drives that could store about as much information as a checkbook register. A graduate of the old People's Computer Co. philosophy of putting computer power in the hands of the people, I told him that every time I consider junking my old TRS-80, I think, "But in its day, it could have launched a Third World space program!" Once ordained in Eastern Europe's original mainframe priesthood, he told me he dislikes Forth and loves languages with libraries.

Draw your own conclusions. Meanwhile, your editor is back at his desk and working on the next couple of issues. But even in the midst of juggling these man-made deadlines, press releases, and various developments, I'm remembering the fragrant sweetgrass and wildflowers, the sound of singers and drums under the full moon, tipis radiant with inner fires, and the age-old lessons of kinship and gratitude.

—Marlin Ouverson

Forth Dimensions
Volume XIV, Number 4
November 1992 December

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

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 $40 per year ($52 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 2154, Oakland, California 94621. Administrative offices: 510-89-FORTH. Fax: 510-535-1295. Advertising sales: 805-946-2272.

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 $40/46/52 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 2154, Oakland, CA 94621."

Letters

Letters to the Editor—and to your fellow readers—are always welcome. Respond to articles, describe your latest projects, ask for input, advise the Forth community, or simply share a recent insight. Code is also welcome, but is optional. Letters may be edited for clarity and length. We want to hear from you!

Visible Words & Ugly Complexity

Dear Marlin:

Thanks to Mike Elola for introducing the topic of graphical interfaces. He mentioned the term "ugly complexity," and it got me thinking again about the perception of complexity.

Three elements contribute to the perception of ugly complexity. The perception of complexity happens when a system forces you to think about more than you are comfortable thinking about. The ugly part of this perception happens when, even after you understand and can use the system, the system still doesn't make sense. The third facet of ugly complexity is finding that you've gained little or no functionality after the struggle to learn the system. Systems like a factory's materials-storage system or the tax laws come to mind.

A given Forth environment seems simple because you don't have to think about very many of its parts at once. You usually have the choice of thinking about only what you can handle. This may be because it is a small Forth system with a relatively small number of words. Even when Forth gets large, it still seems simple because you have a choice about how much you have to deal with at once. The C language, because of its syntax and compilation, doesn't allow nearly as much flexibility concerning what you'll think about, and when.

Give Forth the ability to swallow whole the work of others and make it interactively available...

It's a mistake to think that, because graphical interfaces are written in C or C++, they are by nature complex. Yes, they have a lot of parts and a lot of layers, but they don't have to be complex. It's also a mistake to believe that all graphical interfaces are equal and that any collection of shapes on a screen constitutes a good visible interface. It is important here to distinguish "graphical" from "visible." Graphical simply means that graphics are used. This is a pretty easy thing to do. Visible means that the system is made visible and, hence, more understandable to the user. This is not so easy but, when done right, removes much complexity.

People who believe in command lines don't have to panic here; instead, take a look at the *Macintosh Programmer's Workshop*. It provides a visible means of creating and using command lines. You can get commands working quickly and save them, if you use them a lot, in a smooth, natural fashion. In fact, it works so smoothly that you may not think you're getting much done. This is because you can actually get a lot done without occupying the best parts of your mind with tasks that should be relegated to the lizard brain.

Considering all of this, I think it's a shame to avoid putting a lovely, simple visible interface on such a lovely, simple system as Forth. Creating a visible Forth environment would be easy, because the concept of "word" translates easily into the visible concept of a "box." One could open the box to see what is inside and to manipulate what is there. Stacks have already been pictured in the literature—all that remains is putting in a mechanism to allow the user to point to and grab items on the stack. Visible words and dictionaries are a much better way of distributing functionality than DLLs.

I'm working on these ideas now, and I invite anybody else who is interested to write or call me. Thanks again, Mike.

Sincerely,
Mark Martino
14115 N.E. 78th Court
Redmond, Washington 98052

A Challenge to Standards Warriors

Chuck Eaker says: Hammer your standards-warfare swords into plowshares and figure out how to use them to break new ground by giving Forth the ability to swallow whole the work of others and make it available in the interactive way we all know and love.

Try this. Develop Forth++, which will operate in a Unix environment. Define Forth++ words which take the name of a (preferably C++) library (such as some of the X libraries) and link the library into the Forth++ environment so that a user can interactively list the classes, functions, etc. provided by the library, create instances, execute methods, and generally perform reckless experiments quickly and cheaply in the manner that, for me, is the essence of Forth.

Off-the-shelf class libraries provide incredible leverage but they are stupifyingly complex, and the documentation is enormous but still incomplete. It takes *forever* to create and run a little program that will give you an answer to how this little widget behaves when you do this weird thing with it that isn't mentioned anywhere in the documentation. If I had Forth++ running in another window, I could significantly increase my productivity.

Devise a Forth++ vocabulary and syntax that I could use for interactive development; and a tool that will translate Forth++ to C++, which I can then compile and link the object file to Forth++, so that I can continue development, then translate...

In my opinion, the proposed standard has more than captured the essence of Forth. What Forth needs is a way to

capture other standards. There are lots of common, well-known libraries out there with which tens of thousands of professionals are familiar. They are using them to leverage themselves into positions of power, from which they can develop sophisticated software quickly and cheaply. Forth can never hope to match this achievement on its own.

Chuck Eaker
P.O. Box 8, K-1 3C12
Schenectady, New York 12301

Combsort Revisited

Dear Mr. Ouverson,

After I sent my article, "Combsort in Forth" to you, I experimented with the shrinkage factor (*FD* XIII/4).

I set up speed tests using Comb1 to double-check the effect on Combsort performance of varying the shrinkage factor. The results are given in Figure One. I discovered that if arrays have randomized elements, even a slight deviation from a factor of 1.3 causes the sort speed to suffer. But if the elements are flat or sorted to some degree, a higher factor actually results in a speed increase. This explains the erratic performance found by Box and Lacey with higher factor values. I don't know why a value of 1.3 proves so critical to Combsort performance, so you'll have to accept it as a given.

Walter J. Rottenkolber
P.O. Box 936
Visalia, California 93279

Megasort in Forth

Dear Editor,

The article by Walter J. Rottenkolber (*FD* XIII/4) on Combsort, was very interesting but the table of contents entry was

Figure Two. More readable Megasort.

```
\ MEGASORT FOR EASY READING
: ARRY ( 16 bit array maker )
  ( Size-in-items ) CREATE 2* ALLOT
  ( Index - Addr ) DOES> SWAP 2* + ;
256 ARRY BUCKETS  ( TO PUT COUNTS OF EACH OCCURENCE )
256 ARRY POINTERS  ( LOCATION TO PUT VALUE )
ITEMS ARRY DATATEMP  ( TEMPERARY ARRAY )

: INITBUCKETS ( init BUCKETS )
   [ 0 BUCKETS ] LITERAL
   512 0 FILL ;

: SCANLSB ( ITEMS - )
   ( FOR EACH ITEM PUT ONE COUNT INTO THE CORRECT BUCKET )
   ( ITEMS ) 0 DO
    1 I S@ 255 AND BUCKETS +!
   LOOP ;

: BUCKETS>POINTERS1 ( MAKE POINTERS TO THE START OF EACH PILE )
   0
   256 0 DO
    DUP I POINTERS ! I BUCKETS @ +
   LOOP DROP ;

: REORDERLSB ( ITEMS - )
  ( MOVE THE ITEMS TO THE PILES DEFINED BY POINTERS )
   ( ITEMS ) 0 DO
    I S@
    DUP 255 AND POINTERS DUP >R
    @ DATATEMP !
    1 R> +!
   LOOP ;

: SCANMSB ( ITEMS - )
   ( ITEMS ) 0 DO
    1 I DATATEMP 1 + C@ BUCKETS +!
   LOOP ;

: BUCKETS>POINTERS2
   0
   256 128 DO ( NEGATIVE NUMBERS FIRST )
    DUP I POINTERS ! I BUCKETS @ +
   LOOP
   128 0 DO
    DUP I POINTERS ! I BUCKETS @ +
   LOOP DROP ;

: REORDERMSB ( ITEMS - )
   ( ITEMS ) 0 DO
    I DATATEMP DUP @
    SWAP 1+ C@
    POINTERS DUP >R
    @ S!
    1 R> +!
   LOOP ;

: MEGASORT ( #Items - ) ( Language Nov 87 )
 INITBUCKETS ( init BUCKETS )
DUP SCANLSB BUCKETS>POINTERS1 DUP REORDERLSB
INITBUCKETS
DUP SCANMSB BUCKETS>POINTERS2 REORDERMSB ;
```

Figure One. Combsort shrinkage factors & performance.

	Sort time in seconds							
Factor	Ramp	Slope	Wild	Shuffle	Byte	Flat	Checker	Hump
1.1	---	---	96	---	---	---	---	---
1.2	49	54	61	62	60	50	52	58
1.3	36	41	52	51	48	35	38	47
1.4	29	36	67	65	64	29	71	108
1.5	---	31	---	---	---	---	---	---
1.6	---	31	---	---	---	---	---	---
1.7	20	27	---	---	---	---	---	---
1.8	---	27	---	---	---	---	---	---
1.9	---	27	---	---	---	---	---	---
2.0	15	31	120+	120+	120+	17	120+	120+

Figure Three. Smaller, faster Megasort1.

```
\ MEGASORT1 FOR SPEED AND SIZE
CREATE BKT/PNTR  512 ALLOT
CREATE DATATEMP ITEMS 2* ALLOT

: 0BKT/PNTR ( init BUCKETS )
   BKT/PNTR 512 0 FILL ;

: CNTLSB ( ITEMS - )
   ( ITEMS ) 0 DO
    2 I S@ 255 AND 2* BKT/PNTR + +!
   LOOP ;

: CNT>PNTR1
   DATATEMP BKT/PNTR
   256 0 DO
    DUP @ >R
    OVER OVER ! 2+
    SWAP R> + SWAP
   LOOP 2DROP ;

: LSB ( ITEMS - )
   ( ITEMS ) 0 DO
    I S@
    DUP 255 AND 2* BKT/PNTR +
    DUP >R @ !
    2 R> +!
   LOOP ;

DATATEMP 1+ CONSTANT DATATEMP1

: CNTMSB ( ITEMS - )
   ( ITEMS ) 0 DO
    1 I 2* DATATEMP1 + C@ 2* BKT/PNTR + +!
   LOOP ;

: CNT>PNTR2
   0 BKT/PNTR 256 +
   128 0 DO
    DUP @ >R
    OVER OVER ! 2+
    SWAP R> + SWAP
   LOOP DROP BKT/PNTR
   128 0 DO
    DUP @ >R
    OVER OVER ! 2+
    SWAP R> + SWAP
   LOOP 2DROP ;

: MSB ( ITEMS - )
   ( ITEMS ) 0 DO
    I 2* DATATEMP + DUP @
    SWAP 1+ C@ 2*
    BKT/PNTR + DUP >R
    @ S!
    1 R> +!
   LOOP ;

: MEGASORT1 ( #Items - ) ( Language Nov 87 )
 0BKT/PNTR DUP CNTLSB CNT>PNTR1 DUP LSB
 0BKT/PNTR DUP CNTMSB CNT>PNTR2 MSB ;
```

misleading. "Rumors of my death have been greatly exaggerated." I pulled out my old code and found there was no competition with the Combsort and DVD&KNKR. It did get me thinking that even DVD&KNKR might not be the fastest and I didn't want to be caught with my pants down. I did some research and found yet a better sort—one called Megasort. I found an article describing it in a November 1987 copy of *Language*. Well, here are the results:

Test Condition:
Test set: Challenge of Sorts
CPU: 10MHz '286
Timing: average of ten passes timed by stop watch and corrected for pattern-generation time.
Forth: F-PC 3.34

Sort	*Speed*	*Size*
COMB1	6.94	-
COMB2	7.05	-
DVD&KNKR	.65	6728 (Not counting stack and names)
MEGASORT	.62	3136 (Not counting stack and names)
MEGASORT1	.48	2616 (Not counting stack and names)

The king is dead, long live the king!

I have included the code for both MEGASORT [Figure Two] and MEGASORT1 [Figure Three], since MEGASORT is more readable and MEGASORT1 is "pedal to the metal." It should be noted that DVD&KNKR is about three times faster than Quicksort for 1000 items and MEGASORT1 is four times faster!

Dwight K. Elvey
Santa Cruz, California

A Hardware-Interrupt Handler

Dr. Tim Hendtlass
Melbourne Australia

Interrupts are powerful, but often are not used because a knowledge of assembly language programming and attention to many details is usually required. This paper describes an interrupt service routine compiler that allows interrupt service routines to be written directly in high-level Forth while hiding all the tedious detail. It was originally developed for teaching, so students could concentrate on what they were doing and why they were doing it, rather than be lost in how they were doing it. Subsequently, it has been used in a number of other situations, principally in the area of scientific instrumentation. Although ISRs written in high-level Forth are slightly slower than those written in assembler, high-level ISRs have applications for general purpose use and are far easier to write and debug. The example given is written for F-PC, but it can readily be adapted to other processors and implementations of Forth.

Introduction to Interrupts

In most computing, the timing is set by the processor. The user supplies input on demand when the processor wants it and receives output when the processor is ready to provide it. Timing is not of great interest in these cases, except to make the task run as fast as possible, overall.

Hardware interrupts may occur at any time, even when Forth is not in control...

However, in some situations such as interfacing, the data must be acquired when the world is ready to provide it, and if it is not acquired it is lost forever. In such cases, correct handling of the inconsistent and variable timing imposed by the world is most important. Such programs have no way of knowing something is going to happen before the moment at which occurs. Of course, it is possible for the processor to periodically stop doing its main task and look to see if something has happened, just in case; but the chance of missing an event is very high unless an enormous proportion of the processing time is spent looking at very frequent intervals.

A better way to respond to random events is to use special hardware to inform the processor when an event has occurred. It 'informs' the processor with an electrical signal called an interrupt, applied to a pin on the processor, which triggers the interrupt response mechanism inside the processor. The processor (normally) will immediately suspend the task it is doing, establish exactly which of the possible sources just interrupted it, and take whatever action has been deemed appropriate to handle interrupts from that source. After performing this action, the processor will return to carry on with the task it was doing before the interrupt occurred. By making the processor subservient to special interrupt hardware, the programmer can write a program that gives its full attention to the main task, safe in the knowledge that these external, spontaneous events will be handled quickly, safely, and automatically when they occur. The programs to handle each of the possible interrupts are quite separate pieces of code which transfer activity from the main program to them and back again automatically when an interrupt occurs. Of course, the hardware must be initialized before it can handle an interrupt.

Interrupt Response Mechanism

The processor response mechanism is generally very similar in all processors. First the processor finishes its current instruction and saves the minimum information that will be needed later to resume as if nothing had happened. Then the processor jumps to a pre-established address and starts executing the instructions there. The (usually) short program the processor executes in response to an interrupt is called the *interrupt service routine* (or ISR for short). There are often a number of them, each starting at a different address. These start addresses are known as the *interrupt vectors*. Usually there is one ISR for each possible interrupt source, although it is possible for two or more interrupting sources to trigger the same routine to service all of them. There must be a special instruction at the end of each ISR that causes the processor to rescue the information it saved before going to

Tim Hendtlass obtained his Ph.D. in Ionospheric Physics in 1974 but later switched to Scientific Instrumentation. He is now an Associate Professor responsible for the Scientific Instrumentation major at the Swinburne Institute of Technology. He discovered Forth in about 1980 and since has used it extensively, first for research and later for teaching. He teaches Forth to about 80 students a year, who use it for learning about instrument interfacing and real-time processing. In research, he has used it in diverse fields: from intelligent adaptive technological support for the elderly, to highly distributed industrial data collection, to devices for the measurement of capacitance under adverse conditions. He likes F-PC because it is a full implementation with adequate support for even vague students and because, as it is public domain, he can share it with all interested persons without restriction. He can be contacted by mail at the Physics Department, Swinburne Institute of Technology, P.O.Box 218 Hawthorn Australia 3122; or by phone (61 3 819 8863) or by fax (61 3 818 3645).

the ISR and use this to return to what it was doing when it was interrupted, carrying on as if nothing had happened.

Preparing a processor to receive interrupts involves first putting the interrupt service routine(s) in place in memory, then arranging for each interrupt to cause the processor to find its way to the correct ISR. How this is to be done depends on the processor; in some simple systems, the manufacturer specifies the start addresses of the interrupt service routines for all the possible interrupts. In this case, all that is required is to put the ISRs into memory starting at the pre-specified addresses. More commonly, a table of start addresses of the ISRs is kept in memory. This allows the ISRs to be anywhere in memory, of any length, and most importantly to be quickly changed by just changing the appropriate entry in the table. It also allows one physical interrupt service routine to service more than one interrupt source.

Interrupts on the 80x8x Processor Family

From now on, we will limit this discussion to the 80x8x processor family on which F-PC runs. In this family a table of 256 addresses is kept, each entry consisting of a four-byte address in segment:offset form. Possible interrupt sources are numbered from zero to 255, and identify themselves by that number when they interrupt. When interrupt source zero interrupts, the processor reads the zeroth entry in the table, goes to that address and executes the ISR there. The response to an interrupt from source number one is the same, except the first entry is read, and so on. The table of ISR start addresses is called the *interrupt vector table.*

There are times when an interrupt would be an acute embarrassment, such as when the processor is placing (or changing) interrupt service routines, or when the processor is running a piece of code that is so time critical that even the briefest interruption cannot be tolerated. To allow for these situations, two special instructions control whether the processor will respond to interrupts. The machine-level instruction *set interrupt flag* (STI) allows it to respond, the instruction *clear interrupt flag* (CLI) stops it from responding. There are also non-maskable interrupts (NMI)[1] which are responded to no matter what the state of the interrupt-enable flag. The processor automatically disables further interrupts as it goes to do an ISR, and re-enables them when the final instruction of the ISR, the special instruction IRET, is executed. If it is the intention that a particular ISR itself may be interrupted if a more important (urgent) interrupt occurs, the programmer must re-enable interrupts with an STI as soon as it is safe for another interrupt to be recognised.

Interrupts can be triggered by either external hardware, as described above, or by software command. The assembly language instruction INT 0 will cause interrupt zero to run just as if a hardware interrupt signal had been received from interrupt source zero; and similarly for all other interrupts. This is very useful for testing purposes.

Figure One. Registers which must be correctly reloaded for 'safe' re-entry to Forth.

The data stack pointer	SP
The return stack pointer	BP
The next instruction pointer	ES:SI
The current word pointer	AX
The scratch pad registers	BX, CX, DX, DI
The segment registers	CS, DS, SS
The direction flag	DF

Designing an ISR Compiler for F-PC

It is most important to realise that once an interrupt occurs and is responded to, the processor is running normal machine code, no matter what it was running when the interrupt occurred. So, if we were running Forth, after an interrupt Forth no longer has control. The ISR must at least start out in assembly code.

If a software command causes an interrupt *while Forth is running our program*, the environment the processor is in at the time of the interrupt is known: it will be in Forth. However, hardware-initiated interrupts may occur at any time, even when Forth is temporarily not in control. (Forth seeks service from DOS from time to time when it needs to use the screen, the keyboard, or the disks.) To handle hardware interrupts successfully, we have to preserve all the same registers as for the software-initiated case (because most of the time Forth will be in control), as well as any registers over and above these that DOS might use (just in case). The net result of this is that, to be quite sure, we have to save all registers at the start of our interrupt service routine and restore them all just before we return from processing our interrupt.

When we wish to run our Forth interrupt service routine, we can make no assumptions about the contents of any register (DOS could have changed them temporarily) and must reload all the ones (shown in Figure One) absolutely required by Forth (the scratch ones do not need to be loaded when we go into the ISR, as we will always be going to the start of a Forth word; but they must be restored before we return from our ISR, in case Forth was in control and their contents were important when the interrupt occurred). So our skeleton interrupt service routine looks like:

- assembly code to save all registers
- assembly code to reload all registers as Forth needs them
- assembly code to switch to high-level code
- high-level code to do what the ISR has to do
- high-level code to return to assembly code

1. The most usual type of interrupts which can be switched on or off at will are called maskable interrupts. There are also non-maskable interrupts (NMI) which cannot be turned off *inside the processor*. These are normally reserved for responding to emergency situations, such as power failing, the consequences of which would be so cataclysmic that responding to them would be more important than anything else the processor might be doing. The response mechanism is almost identical to the way the processor responds to maskable interrupts, and the words we develop here will work with either maskable or non-maskable interrupts. The address of the non-maskable interrupt service routine is entry 2 in the interrupt vector table.

For IBM-PC users, non-maskable interrupts can be turned off by hardware *external to the processor*. Indeed, they are turned off at power-up (but turned back on by the BIOS almost immediately). They may be turned on by a program writing 80 hex to I/O port A0 hex or turned off by writing 0 to the same port. This uses hardware provided on the PC motherboard to control a gate which allows or prevents the actual electrical NMI signal reaching the chip. It does not exercise control within the processor as CLI and STI do for maskable interrupts. If the electrical signal for a non-maskable interrupt reaches the processor, no power on earth will prevent the processor from responding to it.

- assembly code to reload all the registers we originally saved
- assembly code instruction to return from interrupt (IRET)

As all but the 'high-level code to do what the ISR has to do' are always the same, we can write them as two words (calling the bit before the high-level code ISRENTRY and the bit after ISREXIT). As a further refinement, we can have a defining word, say INT:, that starts an ISR definition. This will build the list that is the user-supplied, high-level ISR code. The definition termination word, say INT;, would append the high-level (colon) version of ISREXIT automatically as the last item on this list. The run-time behaviour ISR: gives to the ISR it is building is to perform ISRENTRY and then to process the list just as if it were a normal colon definition. Our ISR structure is now:

```
ISR: <name>
high-level Forth words
ISR;
```

This is conceptually neater and encourages programmers to concentrate on what they are trying to do rather than the details of how it is being done.

Figure Two. Source code for the ISR-building words.

```
5 constant STACK-NUMBER                    \ # stacks = nesting depth of ISRs
variable STACK-BASE                        \ place to keep the top of the current stack
100 constant STACK-SIZE                    \ size of one data stack return stack pair
A0 constant RSTACK-OFFSET                  \ depth of data stack (offset to return stack)
create ISR-STACKS                          \ pointer to bottom of the stack of stacks
stack-size stack-number *                  \ number of bytes the stacks will take
allot                                      \ make space for the stacks.
isr-stacks stack-size +                    \ calculate top of first data stack
stack-base !                               \ initialize base pointer

LABEL ISRENTRY

comment:
( stack on entry = pc cs flags n )
( old stack on exit = pc cs flags n ax di bp bx ds )
( new stack on exit = es si old-sp old-ss cx dx )
n is the offset in list space to the list of high-level words to do in this ISR. We
first use the stack we are in when the interrupt occurred to save some information
comment;

PUSH AX PUSH DI PUSH BP
MOV BP, SP                                 \ stack pointer to bp
MOV DI, 6 [BP]                     \ adr of offset to list to process (n) to di
MOV CS: AX, 0 [DI]                 \ get the actual offset (from the code segment)
PUSH BX                                    \ we will also need BX
PUSH DS                                    \ and DS
\ old stack is now pc cs flags n ax di bp bx ds.
\ Register ax contains the actual offset into Forth list space
\ Switch to new stack
MOV BP, SP  MOV DI, SS                     \ old stack pointers to bp and di
MOV BX, CS                                 \ new stack segnment=new code segment
MOV SS, BX
MOV DS, BX                                 \ data seg = stack seg = current code seg
MOV BX, # STACK-BASE                       \ get new stack pointer
MOV SP, 0 [BX]                             \ new stack set up

\ Finish setting up the registers for Forth and
\ saving any registers not already saved

ADD 0 [BX], # STACK-SIZE WORD      \ adjust stack-base lest we get
                                           \ interrupted
PUSH ES PUSH SI                            \ save registers we are going to use
ADD AX, # XSEG @ MOV ES, AX                \ point es to the correct list segment
SUB SI, SI                                 \ clear si (part of Forth program counter)
PUSH BP PUSH DI PUSH CX PUSH DX
MOV BP, SP  SUB BP, # RSTACK-OFFSET        \ Set up new return stack pointer
NEXT                                                \ Start the ISR.
                                           \ New stack now  es si old-sp old-ss cx dx
```

(Continued on next page.)

Implementation of the ISR compiler.

The definitions of ISRENTRY, ISREXIT, ISR:, and ISR; are shown in Figure Two. It is not necessary to understand how they work to use them, but these notes are intended to assist those who are curious or wish to modify them for a different system.

When the interrupt occurs, we do not know where the stacks' pointers used by F-PC point, nor do we know how much room exists on these stacks before we write over something important. Although F-PC has a substantial amount of stack space, other versions—especially those on embedded systems—do not, and the only safe thing is to have a pair of new stacks (one for data, one for return addresses) exclusively for the use of our interrupt. We cannot have only one pair of stacks available if this interrupt may itself be interrupted. For interruptable interrupts, we need as many pairs of stacks available as the maximum depth to which we will allow interrupts to be nested. In short: a stack of pairs of stacks, the depth of which determines the maximum interrupt nesting depth. In Figure Two, this is set arbitrarily at five. On entry to the ISR, a variable STACK-BASE is read to get the initial value of the data stack pointer, then this is incremented by STACK-SIZE so it points to the next stack to use should this interrupt be interrupted. The return stack pointer is initialized to the data stack pointer minus RSTACK-OFFSET. At the time of exit from the ISR, the value of STACK-BASE is decremented by STACK-SIZE. In the interests of speed, no check is made to see that you do not run out of ISR stacks (that is, have interrupts nested too deep).

When we get to ISRENTRY, the stack already contains four items of interest to us. The contents of the instruction pointer, the code segment register, and the flag register were saved automatically by the interrupt-handling hardware built into the processor. The minimum run-time behaviour of CREATE places on the stack the address of the word after the call to the run-time routine. In this case, as for a colon definition, this contains the offset from the start of the list segment to the start of the list of things to do. For a description of the internal structure of F-PC, see [Ting89]. A few more things must be saved to give us some working room before we switch to our interrupt stack. Then the remainder of the things we need to save are placed on this new stack.

When we come to the end of the ISR, we cannot just jump back into what we were doing before the interrupt occurred.

```
END-CODE

CODE ISREXIT

comment:
( old stack on entry = pc cs flags n ax di bp bx ds )
( new stack on entry = es si old-sp old-ss cx dx )
( both stacks empty on exit )
comment;

MOV BX, # STACK-BASE
SUB 0 [BX], # STACK-SIZE WORD                \ adjust stack-base down one level
POP DX POP CX POP AX POP BP
POP SI POP ES                              \ restore registers saved on ISR stack
MOV SP, BP  MOV SS, AX                     \ switch back to the entry stack
POP DS POP BX POP BP POP DI                \ restore all but one register we had there
MOV AL, # 20  OUT # 20 AL                  \ re-enable the Pcs hard. int. controller
POP AX ADD SP, # 2                         \ lose offset to list we processed
IRET                                       \ finished with this interrupt
END-CODE


: ISR:                                       \ Interrupt Service Routine defining word
                                             \ builds the name and list of things to do
create                                       \ build header (in head seg)
xhere paragraph +                  \ justify list pointer to next multiple of 16
dup xdpseg !                       \ save one copy into xdpseg
xseg @ -                           \ calc offset from xseg to where list will start
,                \ place after where the jump to isrentry will be code space
xdp off                    \ set xdp to 0
]                          \ make  a list of the colon words that make up the
                           \ ISR continuing until compiler turned off by ISR:
isrentry                   \ get address of run time routine we will use
last @                     \ get name field address of this definition
name>                      \ move to start of code field
1+                         \ move over the opcode byte (call)
tuck 2+ -                  \ calculate relative offset
swap !                     \ install offset to isrentry as target of call
;


: ISR;
state @ 0=                                   \ check we really compiling
abort" Not compiling an ISR!"
                                             \ abort if word used out of turn
?csp                                         \ check for any stack errors, abort if any
compile ISRexit            \ add special exit word ISR; to the ISR list
[compile] [                \ when encountered will turn off the list compiler
; immediate                \ this word must run when compiling
```

First we must execute ISREXIT to return all the registers exactly as they were when the interrupt occurred. This reclaims everything from the interrupt stack, resets the interrupt stack pointer down one level, switches back to the original stack, reloads all the information we saved there, loses the list offset which is still there but no longer needed, then issues the special command that signifies to the PC hardware interrupt controller that the current interrupt is finished, and finally lets the processor do its normal end-of-interrupt housekeeping.

The remainder of the code in Figure Two defines the words that build ISR type words. The first ISR: marks the start of an ISR definition. It builds the list of things to do in list space, just as the colon defining word : does. However, unlike : which installs NEST as the run-time behaviour, ISR: installs the word ISRENTRY which we just wrote.

The list compiler] will continue to build the list until turned off. ISR;, the word that marks the end of the definition, turns off the list compiler and then adds the special word ISREXIT to the end of the list. When this word is processed, the registers are reloaded and control is returned to whatever was going on before the interrupt.

Convenience Words for Interrupts

To handle a source of interrupts, first one would have to write the interrupt service routine and then install it by putting the address of this ISR in the correct place in the interrupt vector table in the memory region from 0:0 to 0:3FFH. Before you write the address of your new ISR, however, you should note the current service installed for that interrupt (apparently 'unused' interrupts may have a trap service installed). The address currently there should be saved so that the original service can be restored later. Of course, if you are sure you will never want to restore the original service, you can write over it. To assist when using interrupts with F-PC, a number of convenience words are reproduced in Figure Three, based on the file INTERRUP.SEQ contributed to the F-PC package by C.H. Ting. ?INTERRUPT returns the address of the current interrupt service routine; this can be replaced later with RE-INSTALL-INTERRUPT. INSTALL-INTERRUPT is used to write the address of a Forth ISR into a specified position in the vector table. You must never get into the situation where an interrupt vector 'points to' (is the address of) an ISR that no longer exists. Disaster is assured if you do so and this interrupt occurs.

It is sometimes convenient to turn interrupts off and on directly with high-level Forth words. Two trivial Forth words to do just that are also shown in Figure Three, as is an example of a word that triggers a software interrupt so you can test an ISR without the hardware needing to be present.

Example of a High-Level ISR

The example shown in Figure Four produces an interrupt-driven counter which is incremented at a regular rate and can be used as the basis for a host of timing purposes.

External hardware interrupts the processor in the IBM-PC family at a regular rate. As well as producing interrupts used by the BIOS in the PC, this hardware signal triggers interrupt 1CH, which normally is serviced by a 'do nothing' ISR in the BIOS.[2] We may re-vector this interrupt to our own ISR that

2. As far as I know such a routine (literally just IRET) is available in every BIOS. However, the address varies from BIOS to BIOS. The only portable way to install interrupt vectors is to read and save what is there and then put it back when you have finished, not to assume that the vector installed was a vector to the do-nothing routine and that this routine exists at a standard address. For this reason, I do not use or allow my students to use the REMOVE-INTERRUPT word from Dr. Ting's file—it just writes in the address F000:FF53 which may or may not be the address of the do-nothing ISR, depending on the BIOS you are using.

increments a 32-bit counter. The interrupt occurs at 18.2 Hz, so our counter will be incremented approximately once every 55 milliseconds.

We need to install this ISR before it can be used, e.g.:

```
hex
2variable OLD-VECTOR
  \ space to save the original vector
  \ we could save it on stack instead

1C ?interrupt old-vector 2!
  \ read and save old vector

' ticking 1C install-interrupt
  \ install our new vector

decimal
```

A couple of other minor words are needed, one to initialize (zero) the value in the counter, and the other to read and display the current value in the counter. These are also shown in Figure Four.

INIT-TICKS will zero the counter and TICKS? will print the current value in the counter. Despite 1C interrupts occurring at, no doubt, inconvenient times as Forth continues to be used, all continues as it should because TICKING meets the requirements of a good ISR: it is short, fast, and leaves no trace of itself on any stack when it has finished running. When we have finished with our ISR for good, we can restore things as they were before we installed it by typing:

```
hex
old-vector 2@              \ get saved vector
1C re-install-interrupt    \ put it back
decimal
```

Remember that interrupt 1C 'fires' 18 times or so every second. So it must always be vectored to a physically existing ISR. Don't leave F-PC and load another program without replacing the original vector, or the system will crash as the memory image of the ISR code of TICKING get overwritten.

Lean, Mean, Interruptable Interrupts and DOS

Interrupt service routines should be as short and as fast at executing as possible. They should never perform any input or output (for example) if it can be possibly avoided, as both of these operations take considerable time. The idea is to service the interrupt but also to make as small an interruption to the main program as possible. The ISR should do the most time-critical part of the total service and, if there is more service to do, set a flag so that the main program can complete the task when it is convenient. For example, when collecting data samples under interrupts, the ISR should just acquire the value from the input port, put it in a holding buffer, and set a flag so that the main program knows to process the values from the buffer when it is convenient. Using a multitasker in conjunction with flags makes this process particularly simple.

When using F-PC with DOS, there is another reason why you should not make use of any DOS-based input or output. Recall that above we arranged for our interrupts to be themselves interruptable. To achieve this, we arranged to have a number of stacks available for use by the ISR, each ISR automatically using the next one above the last one used. DOS has no such facility. It always uses the same stack for a given function. So if, for example, we are outputting to the screen, DOS will set up a stack for its use at a fixed place. If, part way through this output operation, another interrupt occurs and the new interrupt also goes to output something, DOS will set up a new stack directly on top of the old one. This will cause no trouble for the interrupt that is currently being serviced, but when that is over and the processor goes to finish the interrupted interrupt, the information it needs has been overwritten. Disaster is now but a few pulses of the

Figure Three. Convenient words for use with interrupts (hex entry is assumed).

```
CODE ?INTERRUPT  ( int# -- seg offset )
    POP AX           \ get interrupt number
    PUSH ES
    PUSH BX          \ preserve these registers
    MOV AH, # 35     \ load DOS service number to AX
    INT 21           \ call DOS to do the work.
    MOV DX, ES       \ segment returned in ES
    MOV AX, BX       \ offset returned in BX
    POP BX
    POP ES           \ restore registers we preserved
    2PUSH            \ put answer on the stack
END-CODE

CODE INSTALL-INTERRUPT  ( addr int# -- )
    POP AX           \ get interrupt number to AX
    POP DX           \ and ISR offset address to DX
    PUSH DS          \ preserve DS for later restoration
    MOV AH, # 25     \ we require DOS service number 25 hex
    PUSH CS          \ ISR segment address is in CS
    POP DS           \ so copy it via stack to DS
    INT 21           \ let DOS do the work
    POP DS           \ restore original DS
    NEXT             \ no values to return, just use NEXT
END-CODE

CODE RE-INSTALL-INTERRUPT  ( seg offset int# -- )
    POP AX           \ get interrupt number to AX
    POP DX           \ and ISR offset address to DX
    PUSH DS          \ preserve DS for later restoration
    POP DS           \ and pop ISR segment address to DS
    MOV AH, # 25     \ we require DOS service number 25 hex
    INT 21           \ let DOS do the work
    POP DS           \ restore original DS
    NEXT             \ no values to return, just use NEXT
END-CODE


CODE INT-ON          STI NEXT END-CODE
CODE INT-OFF         CLI NEXT END-CODE

CODE TRIGGER-INT-1C  \ replace 1C by the interrupt
                     \ number you wish to test
    INT 1C NEXT
END-CODE
```

Figure Four. Example of a high-level interrupt service routine plus test words.

```
2variable ticks

: DINC  ( adr -- )
  dup 2@ 0.1 d+ rot 2! ;  \ increment a double variable

: INIT-TICKS  ( -- )
  0 0 ticks 2! ;          \ initialize the counter to zero

: TICKS?  ( -- )
  ticks 2@ ud. ;          \ read and display the counter

ISR:
TICKING ticks dinc ISR;   \ that's it - the whole ISR
```

processor clock away. Avoiding DOS service in our ISRs is the only way to ensure this never occurs.

Extra Info about IBM PC Hardware Interrupts

The information given so far describes how the processor itself handles interrupts. Many computers use extra hardware external to the processor, that provides extra control over interrupts—in particular to exercise various forms of priority control which allow high-priority interrupts to take precedence over lower-priority ones. The IBM PC/XT/AT family is no exception and has one or more 8259A interrupt-priority controller(s), which provides various features at the cost of having to be programmed. A full discussion of this chip is outside the scope of this paper, but the following section should provide enough information to allow use to be made of the interrupt lines on the I/O bus of the IBM PC family of computers. For information about features not discussed here, such as changing the priorities of the various interrupt request signals, the user is referred to the 8259A data sheet.

The I/O bus of the IBM PC and XT provides six lines, called IRQ2 through IRQ7, each of which signals that an interrupt service is required when taken high. Two other lines are are also on the motherboard but are not brought out onto the I/O bus. The electrical signals on these lines have to pass through the interrupt controller chip to get to the processor. The controller decides which, if any, request should be passed on to the processor. It decides this based on the priority of the interrupt (whether this is of high enough priority to be allowed to interrupt what the processor is currently doing) and whether it has been explicitly disallowed from passing on this type of interrupt. Each of the signals from the eight lines may be disabled by writing a 1 to the appropriate bit in a register inside the 8259A. Bit 3 of this register controls line IRQ3, etc. The IBM AT has more IRQ lines on the secondary I/O channel 8259A controller and uses the normal IRQ2 to indicate activity on the secondary 8259A controller IRQ lines.

The eight interrupt request lines on the I/O bus, their normal use, and the interrupt number they are mapped to are listed in Figure Five. Each line may be used by an end user's hardware, although difficulties will be experienced if the normal 'owner' of a line uses it at the same time. If you do install your own interrupt service routine for any of these interrupts, be sure to restore the one normally there when you are done.

An interrupt can be signaled by bringing the relevant IRQ line from the low to the high state. It must be kept in the high state until the interrupt service routine for this interrupt has begun. As initialized by the BIOS, the interrupt controller will not pass a second interrupt signal to the processor until it has been given a signal to do so. This signal is given by the processor writing 20 hex to output port 20 hex. This is automatically done by the code of `ISREXIT` at the end of the ISR, but can also be done as soon as it would be convenient to receive another interrupt. It does not matter if the controller is reset more than once. Do not confuse this signal, which re-enables the external interrupt priority controller chip, with the interrupt enable flag inside the processor. The external interrupt priority controller can stop any hardware interrupt signal from passing on to the processor. The processor interrupt enable flag will stop or allow all maskable interrupts, hardware- or software-triggered.

The mechanism by which the relevant IRQ line was held high until the ISR was started (usually a flip-flop) must be reset by the ISR routine itself as the interrupt-acknowledge signal from the processor is not brought out onto the I/O bus. Thus, the ISR will need to have two extra items in it over and above what it needs to suit the processor and the main ISR task to be done—it needs to reset the interrupt priority controller (automatically done) and it needs to reset the IRQ generating mechanism (left to the programmer).

Figure Five. Interrupt Request Lines on the IBM PC.

IRQ0	Used for system timing applications and is mapped to interrupt 8. Interrupt 8 on completion passes control to interrupt 1C (hex), which is the user timer interrupt and whose vector normally points to a simple IRET. This line does not appear on the I/O channel.
IRQ1	Used for the keyboard and mapped to interrupt vector 9. This line does not appear on the I/O channel.
IRQ2	Reserved in the PC and XT. It is used in the AT family to receive the output of another 8259A, so that a total of 15 individual interrupts can be handled. It is vectored to interrupt number 0A (hex).
IRQ3	Normally used by the secondary asynchronous communications device (COMS2) and mapped to interrupt number 0B (hex).
IRQ4	Normally used by the primary asynchronous communications device (COMS1) and mapped to interrupt number 0C (hex).
IRQ5	Normally used by the fixed (hard) disk and mapped to interrupt number 0D (hex).
IRQ6	Normally used by the diskette (floppy disk) and mapped to interrupt number 0E (hex).
IRQ7	Normally used by the parallel printer (PRN) and mapped to interrupt number 0F (hex).

The 8259A is fairly complex; although it only occupies two output ports, it is programmed by sending information by way of strings of bytes written in carefully controlled sequences to these two ports. To rewrite the contents of the interrupt mask register (the register that determines which interrupts are categorically not to be allowed through), one needs to do more than just write the one byte that controls each of the eight lines. The sequence required is: 13 hex to output port 20 hex, 8 hex to output port 21 hex, 9 hex to output port 21 hex, and finally the interrupt mask to output port 21 hex. The values given here will result in the interrupt mask being changed, but they preserve all the other features as set up by the BIOS at system initialization. See an 8259A data sheet or [Eggebrecht83] for the meaning of each bit and the sequences needed to alter other features.

References

[Ting89] *F-PC Technical Reference Manual*, Offete Enterprises, Inc. 1306 South B Street, San Mateo, California 94402.

[Eggebrecht83] *Interfacing to the IBM Personal Computer* by Lewis C. Eggebrecht, Howard W. Sams & Co.

Principles of Metacompilation

B.J. Rodriguez
Hamilton, Ontario, Canada

F. Creating the Forth Header

Assembly code rarely exists in isolation in a Forth system. Usually, it is part of a Forth "word" (dictionary entry). This requires that some information be prefixed onto the machine code.

1. Use

The Forth word CODE performs two functions: it builds the header for a Forth dictionary entry, then it invokes the assembler. A word of the same name in the "hosting" vocabulary will begin a code word for the Target image.

```
HOST CODE name
```

Starts a Target "code word." Builds a Forth header with the given name in the target image, and invokes the cross-assembler.

Normally, during cross-assembly, the HOST vocabulary (or its ASSEMBLER branch) remains active throughout a cross-assembly. It is not necessary to return to the NATIVE vocabulary. So, once HOST is selected, each code word can begin with simply

```
CODE name
```

Depending on the assembler, it may be necessary to end each code word with ;C or END-CODE.

2. Implementation (screen 75)

This is the first point at which the structure of the Target machine's Forth must be known.

It is not likely that the Target Forth's header structure is the same as the Host Forth's. There is no shortcut; it is necessary to write a word which causes the Host to build a header in the format required by the Target machine.

```
(TCREATE) name
```

Builds a header in the Target image, in the format required by the Target's Forth.

Note once again the use of the T-prefix, rather than just a different vocabulary, to distinguish this word from the native (CREATE). Both will be needed.

Figure Three illustrates the dictionary header for a common fig-Forth model. The "name field" consists of one byte, indicating the name length (0..31) followed by the name text, with the high bit set in both the length byte and the last text byte. The next two bytes are the "link field," a pointer to the name field of the previous definition. The last two bytes are the "code field," pointing to the executable machine code for this word. In the case of a CODE word, the executable code is stored immediately after the code field address ("CFA").

This implementation takes advantage of the fact that *the name field in the Host is stored in exactly the same format.* The work of parsing a name from the input stream, and adding the length byte and the "end bits." (TCREATE) assumes that a Host CREATE has already been performed, and simply copies the name field (with >TCMOVE) to the Target image.

The link field and code field must be explicitly handled for the Target image, since they bear no relation to the Host. Since the link field must have the Target image address of the previous Target definition, the compiler must maintain a LATEST for the Target.

```
HOST LATEST     ( -- a )
```

Returns the Target address of the last definition added to the Target dictionary. (screen 72)

Since this is a fig-Forth model, LATEST is implemented by referencing a pointer to the current vocabulary header. Although the vocabulary header is stored in the Target image, the pointer to it is a variable in the Host. Thus, LATEST is defined as:

```
CURRENT  @  T@
```

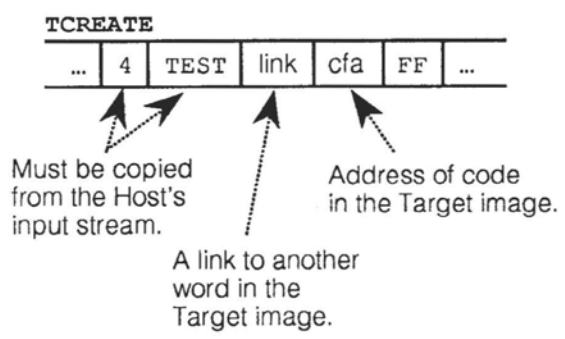
where CURRENT is the name of the pointer, @ fetches the contents of the pointer, and T@ then gets the last-entry information from this address in the Target image.

To maintain the fig-Forth vocabulary structure, the following pointers must be kept. They are defined in the HOST vocabulary, and *are stored in the Host memory space.*

```
HOST CURRENT
```

Holds the pointer to the vocabulary header, for the vocabulary

Figure Three. The dictionary—creating the header.



The host must keep a LATEST pointer for the image!

where new definitions are "currently" being added.

`HOST CONTEXT`
Holds the pointer to the vocabulary header, for the vocabulary which is to be searched for references to already-defined words.

`HOST VOC-LINK`
Holds the pointer to the vocabulary header, for the most recently defined vocabulary.

3. Issues

a) Direct-Threaded Code

The fig-Forth implementation for the Zilog Super8 uses Direct-Threaded Code, rather than the Indirect-Threaded Code more commonly seen in Forth. Direct-Threaded Code does not use a code field pointer; instead, the executable machine code for each word directly follows the link field.

The relative merits of direct vs. indirect threading are a hotly debated topic in Forth circles. In this case, the fact the Super8 CPU includes instruction-level support for DTC was the deciding factor.

The impact on the Image Compiler is that, for `CODE` words, nothing need be compiled by `(TCREATE)` after the link field—the assembler is invoked immediately. For high-level and defined words which use a common machine language routine for all the words in a class, a subroutine call must be compiled after the link field. In practice, `(TCREATE)` always compiles the subroutine call, and `CODE` "removes" this unnecessary call by backing up the dictionary pointer three bytes.

b) Word alignment

Some machines (notably the PDP-11 and the 68000) require that 16-bit values, such as addresses, be word-aligned in memory. This is commonly ensured by word-aligning the definitions, and the link and code address fields.

This must be accomplished within `(TCREATE)`. Normally, `(TCREATE)` will begin with a word named something like `ALIGN`, which forces the Target Dictionary Pointer to an even boundary. Then, if the combination of length byte and name text is an odd length, a null will be appended to the name to make it even. (Whether or not this null is included in the length byte value is problematical.)

c) Packed name fields

Occasionally, clever schemes are devised to speed up dictionary searches by compressing or packing the name information. One PDP-11 implementation [3] packed four characters of name, the length, and the link into two 16-bit words.

All of this, if desired, is the responsibility of `(TCREATE)`.

d) Different linking methods

Other linking methods than the simple, last-to-first, singly linked list are possible. `(TCREATE)` is the word most affected by these.

Links can be stored in forms other than addresses (as in [3]).

Several versions of Forth use multiple dictionary threads to speed the sequential search. Which thread to search for any given name is decided by performing a hashing function on the name. (This has repercussions in vocabulary structure as well, as will be seen shortly.)

e) Separated headers

It is becoming increasingly common for the header information—specifically, the length, name, and link—to be stored in a separate region of memory. On the IBM PC, for example, a separate 64K segment can be devoted exclusively to dictionary headers, thus freeing more space in the 64K "program" segment.

4. Alternatives

a) Re-scanning the name text.

At least one metacompiler creates the name field in the Target, not by copying a name field from the Host machine, but by rescanning the input text. The name is parsed with `WORD`, and then it and its length are copied to the Target image. The text input pointer is then backed up to the start of the name so that the Host's `CREATE` can parse the input normally. (The need for parsing the name twice will become evident shortly.)

G. Searching the Target Dictionary (mirror vocabularies)

The reason Forth words have this header information is so they may be found by name later. This is the core of the "high-level" Forth compilation process: each word in a new definition is searched in the "dictionary" and the address of its executable code is compiled.

Obviously, a metacompiler must be able similarly to find

words in the Target's dictionary.

1. Usage

Words created in the Target image are accessed by name, just like any other Forth words.

If the Host is in the "compiling" state, Target words are compiled into the Target image. (More on this later.)

If the Host is in the "executing" state, Target words generate an error. The words being created in the Target image are not executable by the Host. (Chances are, they are for a different CPU entirely.)

It will be seen later that, under some circumstances, a word defined in the Target may also have an "executing" behavior in the Host.

2. Implementation

Every word defined in the Target image has a corresponding word, of the same name, defined in the Host system. These words in the Host system are called "mirror" words.

The metacompiler never needs to search through the Target image. The Host's own, ordinary search logic is sufficient to find the mirror word in the Host's dictionary. Each mirror word identifies where its counterpart is located in the Target image. (This eliminates the need for the metacompiler to have a `TFIND`—a non-trivial problem).

Figure Four shows the relationship between the Target dictionary and the Host dictionary. This illustrates a kernel word, `LIT`, as it appears in the dictionary being built in the Target image, and in the Host dictionary.

It is likely that many words defined in the Target will have the same name as important words in the Host. (If the metacompiler is creating a new Forth kernel, this is certain.) To avoid these name conflicts, and to allow words to be found unambiguously, all of the mirror words are kept in yet another vocabulary, called `TARGET`, as shown in Figure Four.

It may well happen that, in the course of writing a metacompiled application, the Forth programmer desires to create vocabularies. Vocabularies are commonly used in Forth to distinguish duplicate names, to control the search order, or to "modularize" the program. The metacompiler must, therefore, duplicate these effects.

Fortunately, with a tree-structured vocabulary system (such as in the fig-Forth model), a tree of any complexity can be represented as a branch of another tree.

This means that all the branching vocabularies in the Target image can be made to correspond exactly with branches from the `TARGET` vocabulary in the Host dictionary. (Figure Five.) And, as long as the Host is in the corresponding vocabulary, it will have exactly the same search order as the Target.

Figure Five shows all the vocabularies likely

Figure Four. The dictionary—searching.

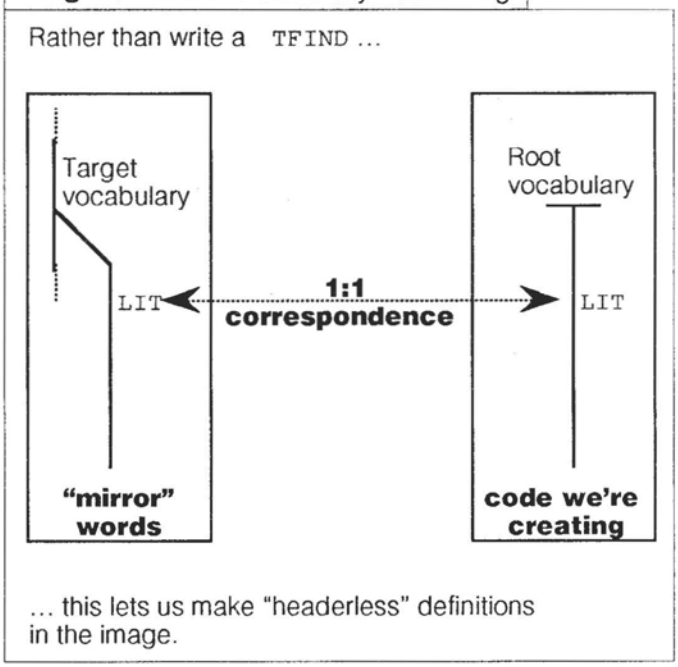
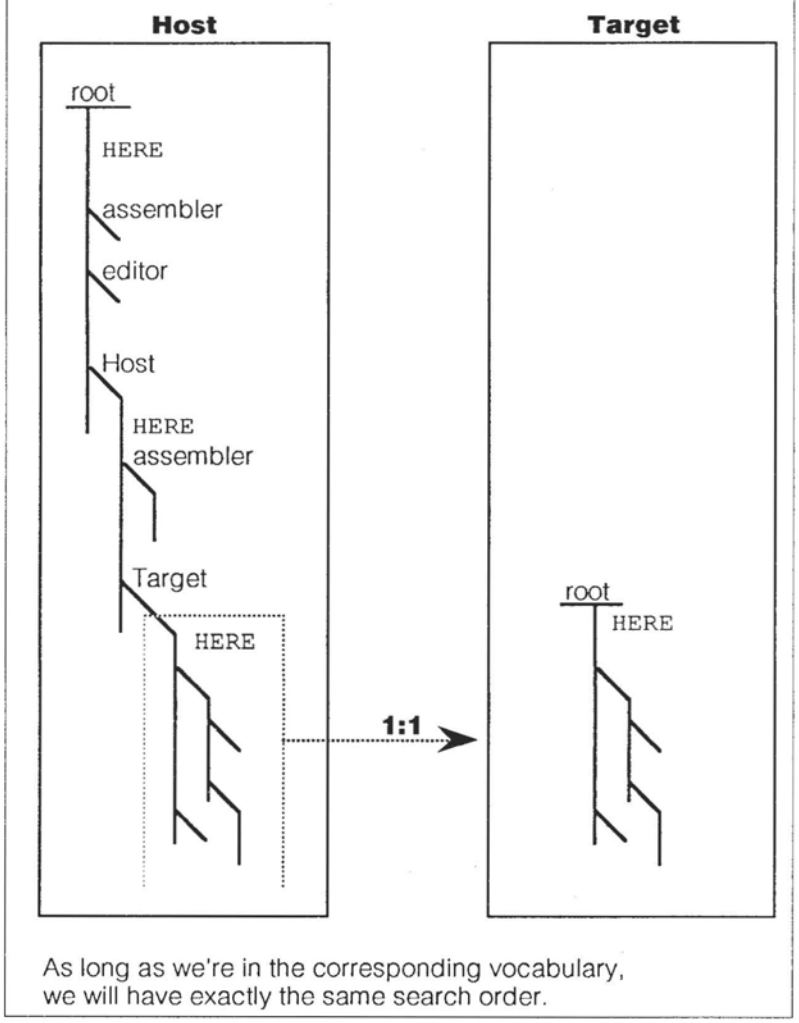


Figure Five. The dictionary—vocabularies.



to be present in the Image Compiler.

"root" FORTH
The basic vocabulary of the Host's Forth system.

"root" ASSEMBLER
The vocabulary which holds the Host's resident assembler. (On the IBM PC, an 8086 assembler.)

"root" EDITOR
The vocabulary which hold the Host's screen editor.

HOST
All of the Image Compiler is contained within this vocabulary and its branches.

HOST ASSEMBLER
The vocabulary which holds the cross-assembler for the Target. (In this example, a Super8 assembler.)

HOST TARGET
All the mirror words created during metacompilation are contained in this vocabulary and its branches.

Observe that there are three words named HERE in Figure Five:

"root" HERE
Returns the Dictionary Pointer of the Host, i.e., where compilation will occur in the Host (if new definitions are added to the Host dictionary).

HOST HERE
Returns the Dictionary Pointer of the Target image; i.e., where compilation will occur in the Target.

TARGET HERE
A mirror word. This example presumes that a Forth kernel is being compiled for the Target machine. All Forth kernels have a word HERE. So, this word points to the Super8 version of HERE in the Target image.

(As an extreme example, it has happened that five different words called I were defined—in the Host kernel, the editor, the resident assembler, the cross-assembler, and the mirror word of the Target kernel.)

Target words are defined with the "host environment" word

HOST CREATE name
Builds a header in the Target image, and a mirror word in the Host dictionary which points to the new word in the Target image.

This word uses (TCREATE) to build the header in the Target image, and the ordinary, "native" Forth <BUILDS to build the header in the Host system for the mirror word. It then adds the Target image address to the mirror word.

This is why it is necessary to use the name of the new word twice.

The resulting mirror word for the LIT example is shown in Figure Six. This data structure appears in the Host's dictionary as an entry in the TARGET vocabulary. The code address field points to machine code, in the Host, which will be executed by the Host when this word is referenced. The address of the corresponding word in the Target image is stored as one of the two data fields following. (The first data field, shown shaded in Figure Six, will be used later.)

The Image Compiler builds the Host header first. It then copies the name field from that header—with adjustments, if necessary—to the Target image.

3. Issues

a) Headerless code

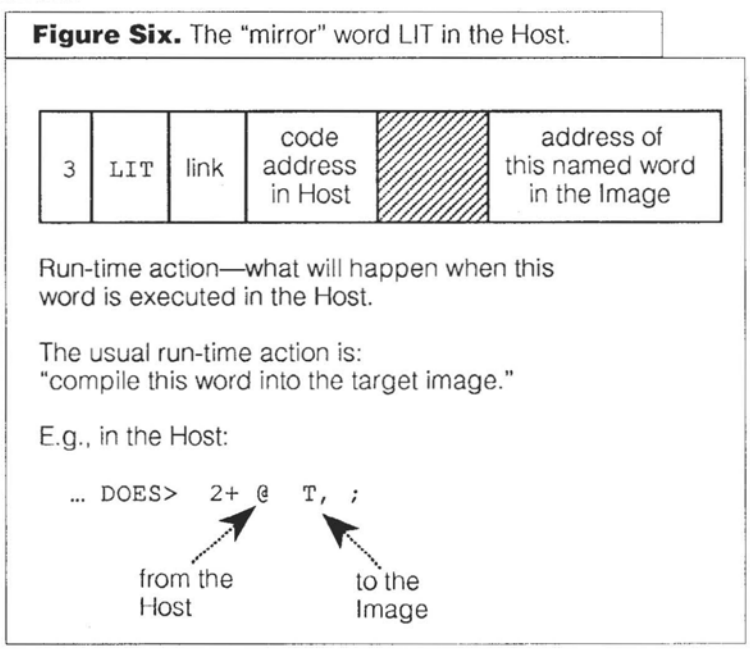
Since the metacompiler always finds words in the Target by searching the Host dictionary, it would seem that the headers in the Target image are dispensable.

They may be, if the final metacompiled application will never need to do a dictionary search. This is likely to be the case in, say, a microwave oven. Such an embedded program is likely to benefit from the memory savings achieved by eliminating the headers from the Target image.

If, on the other hand, the metacompiled application will be using the Forth interpreter—for example, if a new Forth kernel is being compiled—then the headers must be retained. It may still be possible to delete the headers from certain words; this is a popular means to protect "internal" words which should never be directly used by the Forth programmer.

The Image Compiler includes a flag variable ?HEADS which is tested in (TCREATE) to disable the code

Figure Six. The "mirror" word LIT in the Host.



which builds the Target image header. It is not sufficient to simply skip (TCREATE), since it also builds the code field—which is always required, headers or no.

b) Different vocabulary structures
Not all Forths use tree-structured vocabularies. polyFORTH, for example, uses eight parallel vocabularies. The current vocabulary is hashed with the name of a word to direct searches to one of eight threads. [9]

Other Forth systems define vocabularies in a hierarchy, but do not cause the vocabularies to chain together as in the fig-Forth model. Each vocabulary is "sealed."

Some Forths (including fig-Forth) search both the CONTEXT and CURRENT vocabularies. Others search only CONTEXT. Still others support a stack or list of vocabularies which are searched in a defined sequence. [9,10]

These variations do not pose a problem when creating the Target image; they are handled by changing the linking logic of (TCREATE). The problem is ensuring that the search order through the mirror words—which use the Host's vocabulary scheme—is the same as the eventual search order in the Target.

The current Image Compiler ignores the problem completely, assuming either that the Target Forth will use a vocabulary structure analogous to the Host machine's, or that the finer subtleties of the search order are not important, as long as the CONTEXT vocabulary is searched first. These assumptions seem to hold true for most applications.

4. Alternatives

a) Single vocabulary compilers
Some metacompilers provide no support for multiple vocabularies. This is adequate for Forth kernels (which use only one vocabulary), but is a handicap in larger applications.

b) Differing name lengths
Some metacompilers allow the length of the name in the Target dictionary to differ from the length of the name used in the Host's "mirror" words. This seems to offer no advantage, and can lead to quite a bit of confusion.

H. Compiling a Colon Definition

The implementation described so far is sufficient to build a Forth dictionary of CODE words for the Target machine. The real power of Forth, however, lies in its ability to use existing words to define new words. These are the high-level "colon" definitions.

1. Use

A colon definition in the Image Compiler looks exactly the same as in "normal" Forth:

```
: name    word  word  ...  word ;
```

This will build a colon definition name in the Target image. All of the "words" are presumed to already have been defined in the Target.

The Image Compiler works with some subtle differences from the normal Forth compiler, though:

a) The word : (colon) does not switch the Host's text interpreter to "compiling" state. It remains in "executing" state.

b) All of `word word ... word` will execute.

c) Each word, when it executes, will compile itself into the Target image.

This technique was described by Laxen [5].

2. Implementation

Each definition in the Target dictionary can be used in the construction of new Forth words in the Target image. (Compiler directives are a special case, to be discussed shortly.)

One approach would be to give the Host's Forth interpreter three states—execute, compile into the Host, compile into the Target. This, however, requires surgery on the Host and complicates the interpreter.

Instead, the function of "compiling into the Target" is achieved by executing words in the Host. These are words in the Host which correspond to the definitions in the Target image—in other words, the "mirror" words.

Each mirror word in the Host belongs to a "class" of words which share the same run-time action: When executed, compile the address of the corresponding Target word, into the Target image. Since the address of the Target word is one of the parameters stored in the Host in the mirror word, this action is represented simply:

```
@  T,
```

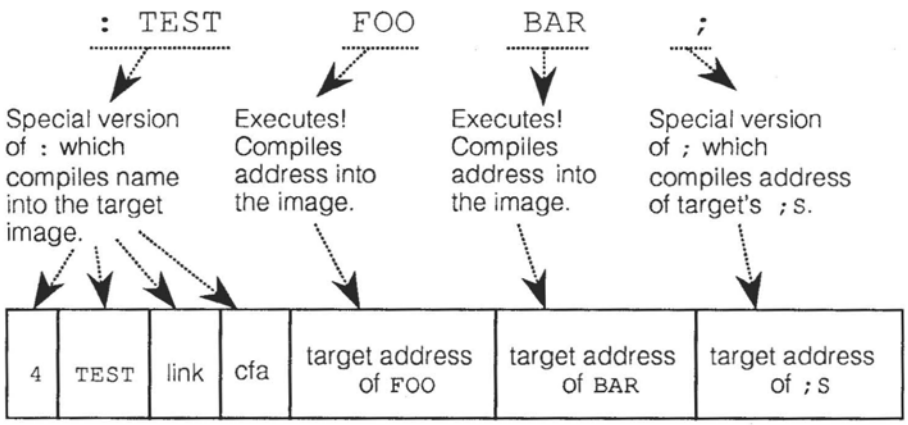
In this implementation a 2+ is prefixed, since the Target word's address is stored in the second word of the parameter field.

All mirror words are created by the HOST version of CREATE. The "self-compiling" action is attached to all of the mirror words by the DOES> clause in CREATE. (Screen 76; also shown in Figure Six.)

This leaves the problem of beginning and ending a colon definition in the Target, i.e., : and ;. To understand these it is best to look at Figure Seven and focus on the First Rule of Metacompiler Design: *Always keep in mind what the result should look like!*

The metacompiler must have a special version of : which constructs a header in the Target image. As shown in Figure Seven, this header must contain the name length, name text, link field, and the code address for a colon definition. This

Figure Seven. Compiling a colon definition.



The entire definition is done in the host's "executing" state. FOO and BAR are "self-compiling" words.

Rule #1 of Metacompiler Design:
Always keep in mind what you want the result to look like!

is the address of machine code in the Target image, which will invoke the Target's colon interpreter. (Strictly speaking, the action is to "nest" the Forth inner interpreter.)

The first three fields are built by the metacompiler's CREATE, which also builds a mirror word in the Host for this new colon definition. The code address can be simply stored in the Target image by T! if the address of this code in the Target is known. For the time being, it will be assumed that this machine code has already been assembled at a known location in the Target image.

(In fact, this code is part of a DOES> clause in the Target's definition of :. The "patching" of Target CFAs by DOES> clauses will be discussed later.)

The job of the metacompiler's ; is much simpler. It must simply compile the address of the Target's ;S word, the run-time routine for ;. ;S is a CODE definition in the Target.

A subtle point is illustrated here. Some parts of the metacompiler— : and ; —must know addresses of certain routines in the Target image. The process of creating the metacompiler and the process of creating the Target image are to some extent "intertwingled." The Image Compiler takes the expedient of first defining the cross-assembler, then the CODE words in the Target, then the rest of the metacompiler.

3. Alternatives

a) Metacompiling by INTERPRET

Of course, "ordinary" Forth does not have words which compile themselves. It is the responsibility of the text interpreter (INTERPRET or]) to "compile the word's address into the dictionary."

The metacompiler could work in the same way. A "metacompiling" text interpreter could be written. It would compile each address—as obtained from the mirror word—into the Target dictionary. (As will be seen shortly, it is necessary to redefine the interpreter loop anyway.)

The advantage of self-compiling words is that their action is somewhat more obvious than a code fragment buried inside INTERPRET. Also, the philosophy of "all words execute" allows quite a bit of flexibility, and some useful "tricks." This will become apparent later.

b) T-prefix naming

The notion of two words named : is confusing to some, even when they can be distinguished by vocabulary. Some systems have used T: to invoke the metacompiler.

While perfectly valid, this is not in keeping with the stated goal of minimizing the differences between "normal" and metacompiled Forth. Many applications will be debugged in a "normal" (resident) Forth environment, and then moved to a metacompiler for optimization and PROM-ing. Consider the amount of editing required to convert every : to a T:!

c) Reverse-patching Target code addresses

Some metacompilers are loaded as a complete unit, before any of the Target code is begun. As noted above, the metacompiler requires the location of certain Target machine code. These conflicting demands are resolved by defining Forth variables within the metacompiler to hold these special addresses. It is necessary to store the correct values in these variables before the metacompiler attempts to use them! (This technique is will be used, for a different Target routine, later in the Image Compiler.)

I. The Problem of Numbers

In addition to previously defined words, numbers may be used to construct a high-level Forth definition.

1. Use

Numbers may be freely intermixed with Forth words in a colon definition:

```
: name    word   1234   word   5678 ;
```

Forth's action on parsing a word from the input stream is: first, check to see if it is an already-defined word. If not, then check if it is a number in the current base. If not, it is an error. The metacompiler works the same way.

2. Implementation

Remembering again the First Rule of Metacompiler Design: a number is compiled as two cells in a Forth definition. The first cell is the address of an executable CODE word, frequently called LIT. The second cell is the number itself, which will not be executed.

The action of LIT when executed will be to fetch the next cell—the number—from the instruction stream, and put it on Forth's stack.

Obviously, the metacompiler must perform similar actions: on encountering a number, compile the address of the Target's LIT into the Target image. Then compile the number itself into the Target image.

The problem lies in how the Host system handles numbers. Unlike Forth words, whose compile-time and run-time actions can be changed, the action for numbers is fixed in the text interpreter, INTERPRET. This action cannot be changed without altering the kernel, which is "off-limits."

Fortunately, the new compile-time action for numbers is only required within the metacompiler. It is perfectly orthodox to redefine the text interpreter before defining the metacompiler. The metacompiler will use the latest defined version, which can have any desired behavior.

Observe that only the "metacompiling" action for numbers need be changed. When a "compiling" interpreter is defined separately from the "executing" interpreter, it is usually made part of the word]. (] means "enter the compiling state" in all Forth systems; whether this enters an interpreter loop or merely sets a flag is system-dependent.) The word : always uses] to enter the compiling state.

So, a "metacompiling"] is created, which is used by the "metacompiling" :. The Host still remains in the "executing" state, and mirror words are still searched and executed in the Host. Only the handling of numbers is different.

(In the next section,] will need to affect the STATE flag, as well.)

Another problem: the metacompiler needs to know the location of the Target's LIT, so that the number-compiling code can know what to compile. Once again, part of the Target code must be assembled before the metacompiler can be completed. In this case, however, the Image Compiler uses an internal variable, *LIT*, to hold this magic Target address. The programmer must store the address of the Target LIT in *LIT*, before attempting to compile any in-line numbers!

3. Issues

a) Double precision

The Forth interpreter recognizes any integer containing a decimal point as a double-precision number.

Usually, double-precision numbers are compiled in-line as two single-precision numbers, with the low cell first. When this sequence is later executed, the two single-precision values will be stacked one after the other to make a double-precision value, with the high cell on top of the stack.

The Image Compiler's number-handling logic (TLITERAL) examines the value of DPL—left by NUMBER—to identify a double-precision number. It then compiles one or two single-precision values, as required.

b) Floating point and other literal values

Similar extensions can be employed to recognize floating-point numbers, and other in-line literal data types. Since Forth's NUMBER provides no mechanism to recognize these, NUMBER must be redefined.

Fortunately, like the text interpreter, a new NUMBER will replace the old, wherever it is used by the metacompiler.

4. Alternatives

a) Redefining INTERPRET instead of]

There are two schools of thought in the Forth community, on how to handle the compiling "state" in Forth.

1) The first school, exemplified by fig-Forth, uses a single text interpreter loop, INTERPRET, and a state flag, STATE. INTERPRET is made "state-smart." When it parses a word from the input stream, it may either compile or execute that word, depending on the value of STATE.

2) The second school, exemplified by polyFORTH and F83, uses two interpreter loops. The "executing" interpreter is INTERPRET and the "compiling" interpreter is] . There is no need for a STATE flag, since the compile vs. execute action is determined by which loop is in progress.

This is not the place for philosophical debates; suffice it to say that either approach can be used within the metacompiler. The former requires the metacompiler to redefine INTERPRET. The latter requires] to be redefined.

Note that the metacompiler needn't use the same technique as the Host machine's Forth. For example, the Image Compiler uses approach (2), while running on a fig-Forth system that uses (1). (Perhaps a minor advantage can be claimed; if the Host Forth ABORTs, it will restart the "native" INTERPRET, which is the same execution interpreter used by the metacompiler under approach (2).)

It is expedient—as will be seen later in this series—to maintain a STATE flag for the metacompiler, regardless.

(Article continues in the next issue. Code begins on page 22.)

FIG
MAIL ORDER FORM

HOW TO USE THIS FORM: Please enter your order on the back page of this form and send with your payment to the Forth Interest Group. All items have one price and a weight marked with # sign. Enter weight on order form and calculate shipping based on location and delivery method.

NEW

"Were Sure You Wanted To Know..."

Forth Dimensions, **Article Reference** 151 - $4 0#
★ An index of Forth articles, by keyword, from *Forth Dimensions* Volumes 1–13 (1978–92).

FORML, Article Reference 152 - $4 0#
★ An index of Forth articles by keyword, author, and date from the FORML Conference Proceedings (1980–91).

FORTH DIMENSIONS BACK VOLUMES

A volume consists of the six issues from the volume year (May–April)

Volume 1 Forth Dimensions (1979–80) 101 - $15 1#
Last 50 Introduction to FIG, threaded code, TO variables. fig-Forth.

Volume 3 Forth Dimensions (1981–82) 103 - $15 1#
Last 10 Forth-79 Standard, Stacks, HEX, database, music, memory management, high-level interrupts, string stack, BASIC compiler, recursion, 8080 assembler.

Volume 6 Forth Dimensions (1984–85) 106 - $15 2#
Last 100 Interactive editors, anonymous variables, list handling, integer solutions, control structures, debugging techniques, recursion, semaphores, simple I/O words, Quicksort, high-level packet communications, China FORML.

Volume 7 Forth Dimensions (1985–86) 107 - $20 2#
Last 100 Generic sort, Forth spreadsheet, control structures, pseudo-interrupts, number editing, Atari Forth, pretty printing, code modules, universal stack word, polynomial evaluation, F83 strings.

Volume 8 Forth Dimensions (1986–87) 108 - $20 2#
Last 100 Interrupt-driven serial input, data-base functions, TI 99/A, XMODEM, on-line documentation, dual-CFAs, random numbers, arrays, file query, Batcher's sort, screenless Forth, classes in Forth, Bresenham line-drawing algorithm, unsigned division, DOS file I/O.

Volume 9 Forth Dimensions (1987–88) 109 - $20 2#
Last 100 Fractal landscapes, stack error checking, perpetual date routines, headless compiler, execution security, ANS-Forth meeting, computer-aided instruction, local variables, transcendental functions, education, relocatable Forth for 68000.

Volume 10 Forth Dimensions (1988–89) 110 - $20 2#
Last 100 dBase file access, string handling, local variables, data structures, object-oriented Forth, linear automata, stand-alone applications, 8250 drivers, serial data compression.

Volume 11 Forth Dimensions (1989–90) 111 - $20 2#
Last 100 Local variables, graphic filling algorithms, 80286 extended memory, expert systems, quaternion rotation calculation, multiprocessor Forth, double-entry bookkeeping, binary table search, phase-angle differential analyzer, sort contest.

Volume 12 Forth Dimensions (1990–91) 112 - $20 2#
Last 100 Floored division, stack variables, embedded control, Atari Forth, optimizing compiler, dynamic memory allocation, smart RAM, extended-precision math, interrupt handling, neural nets, Soviet Forth, arrays, metacompilation.

FORML CONFERENCE PROCEEDINGS

FORML (Forth Modification Laboratory) is an educational forum for sharing and discussing new or unproven proposals intended to benefit Forth, and is an educational forum for discussion of the technical aspects of applications in Forth. Proceedings are a compilation of the papers and abstracts presented at the annual conference. FORML is part of the Forth Interest Group.

1980 FORML PROCEEDINGS 310 - $30 2#
Address binding, dynamic memory allocation, local variables, concurrency, binary absolute & relocatable loader, LISP, how to manage Forth projects, n-level file system, documenting Forth, Forth structures, Forth strings. *231 pgs* **Last 10**

1981 FORML PROCEEDINGS 311 - $45 4#
CODE-less Forth machine, quadruple-precision arithmetic, overlays, executable vocabulary stack, data typing in Forth, vectored data structures, using Forth in a classroom, pyramid files, BASIC, LOGO, automatic cueing language for multimedia, NEXOS—a ROM-based multitasking operating system. *655 pgs* **Last 50**

1982 FORML PROCEEDINGS 312 - $30 4#
Rockwell Forth processor, virtual execution, 32-bit Forth, ONLY for vocabularies, non-IMMEDIATE looping words, number-input wordset, I/O vectoring, recursive data structures, programmable-logic compiler. *295 pgs* **Last 100**

1983 FORML PROCEEDINGS 313 - $30 2#
Non-Von Neuman machines, Forth instruction set, Chinese Forth, F83, compiler & interpreter co-routines, log & exponential function, rational arithmetic, transcendental functions in variable-precision Forth, portable file-system interface, Forth coding conventions, expert systems. *352 pgs* **Last 100**

1984 FORML PROCEEDINGS 314 - $30 2#
Forth expert systems, consequent-reasoning inference engine, Zen floating point, portable graphics wordset, 32-bit Forth, HP71B Forth, NEON—object-oriented programming, decompiler design, arrays and stack variables. *378 pgs* **Last 100**

1986 FORML PROCEEDINGS 316 - $30 2#
Threading techniques, Prolog, VLSI Forth microprocessor, natural-language interface, expert system shell, inference engine, multiple-inheritance system, automatic programming environment. *323 pgs* **Last 100**

1987 FORML PROCEEDINGS 317 - $40 3#
Includes papers from '87 euroFORML Conference. 32-bit Forth, neural networks, control structures, AI, optimizing compilers, hypertext, field and record structures, CAD command language, object-oriented lists, trainable neural nets, expert systems. *463 pgs* **Last 50**

1988 FORML PROCEEDINGS 318 - $40 2#
Includes 1988 Australian FORML, Human interfaces, simple robotics kernel, MODUL Forth, parallel processing, programmable controllers, Prolog, simulations, language topics, hardware, Wil's workings & Ting's philosophy, Forth hardware applications, ANS Forth session, future of Forth in AI applications. *310 pgs* **Last 100**

1989 FORML PROCEEDINGS 319 - $40 3#
Includes papers from '89 euroFORML. Pascal to Forth, extensible optimizer for compiling, 3D measurement with object-oriented Forth, CRC polynomials, F-PC, Harris C cross-compiler, modular approach to robotic control, RTX recompiler for on-line maintenance, modules, trainable neural nets. *433 pgs* **Last 50**

1990 FORML PROCEEDINGS 320 - $40 3#
Forth in industry, communications monitor, 6805 development. 3-key keyboard, documentation techniques, object-oriented programming, simplest Forth decompiler, error recovery, stack operations, process control event management, control structure analysis, systems design course, group theory using Forth. *441 pgs* **Last 50**

★ - These are your most up-to-date indexes for back issues of *Forth Dimensions* and the FORML proceedings.

Fax your orders 510-535-1295

1991 FORML PROCEEDINGS 320 - $50 3#
NEW Includes 1991 FORML, Asilomar, euroFORML '91, Czechoslovakia and 1991 China FORML, Shanghai. Differential File Comparison, LINDA on a Simulated Network, QS2: RISCing it all, A threaded Microprogram Machine, Forth in Networking, Forth in the Soviet Union, FOSM: A FOrth String Matcher, VGA Graphics and 3-D Animation, Forth and TSR, Forth CAE System, Applying Forth to Electric Discharge Machining, MCS96-FORTH Single Chip Computer. *500 pgs*

BOOKS ABOUT FORTH

ALL ABOUT FORTH, 3rd ed., June 1990, Glen B. Haydon 201 - $90 4#
Annotated glossary of most Forth words in common usage, including Forth-79, Forth-83, F-PC, MVP-Forth. Implementation examples in high-level Forth and/or 8086/88 assembler. Useful commentary given for each entry. *504 pgs*

THE COMPLETE FORTH, Alan Winfield 210 - $14 1#
A comprehensive introduction, including problems with answers (Forth-79). *131 pgs*

eFORTH IMPLEMENTATION GUIDE, C.H. Ting 215 - $25 1#
eForth is the name of a Forth model designed to be portable to a large number of the newer, more powerful processors available now and becoming available in the near future. *54 pgs* (w/disk)

F83 SOURCE, Henry Laxen & Michael Perry 217 - $20 2#
A complete listing of F83, including source and shadow screens. Includes introduction on getting started. *208 pgs*

FORTH: A TEXT AND REFERENCE 219 - $31 2#
Mahlon G. Kelly & Nicholas Spies
A textbook approach to Forth, with comprehensive references to MMS-FORTH and the '79 and '83 Forth standards. *487 pgs*

THE FIRST COURSE, C.H. Ting 223 - $25 1#
NEW This tutorial's goal is to expose you to the very minimum set of Forth instructions so that you can start to use Forth to solve practical problems in the shortest possible time. "... This tutorial was developed to complement *The Forth Course* which skims too fast on the elementary Forth instructions and dives too quickly in the advanced topics in a upper level college microcomputer laboratory. ..." A running F-PC Forth system would be very useful. *44 pgs*

THE FORTH COURSE, Richard E. Haskell 225 - $25 1#
This set of 11 lessons, called *The Forth Course*, is designed to make it easy for you to learn Forth. The material was developed over several years of teaching Forth as part of a senior/graduate course in design of embedded software computer systems at Oakland University in Rochester, Michigan. *156 pgs* (w/disk)

FORTH ENCYCLOPEDIA, Mitch Derick & Linda Baker 220 - $30 2#
A detailed look at each fig-Forth instruction. *327 pgs*

FORTH NOTEBOOK, Dr. C.H. Ting 232 - $25 2#
Good examples and applications. Great learning aid. poly-FORTH is the dialect used. Some conversion advice is included. Code is well documented. 286 pgs

FORTH NOTEBOOK II, Dr. C.H. Ting 232a - $25 2#
Collection of research papers on various topics, such as image processing, parallel processing, and miscellaneous applications. *237 pgs*

F-PC USERS MANUAL (2nd ed., V3.5) 350 - $20 1#
Users manual to the public-domain Forth system optimized for IBM PC/XT/AT computers. A fat, fast system with many tools. *143 pgs*

F-PC TECHNICAL REFERENCE MANUAL 351 - $30 2#
A must if you need to know the inner workings of F-PC. *269 pgs*

INSIDE F-83, Dr. C.H. Ting 235 - $25 2#
Invaluable for those using F-83. *226 pgs*

LIBRARY OF FORTH ROUTINES AND UTILITIES, James D. Terry 237 - $23 2#
Comprehensive collection of professional quality computer code for Forth; offers routines that can be put to use in almost any Forth application, including expert systems and natural-language interfaces. *374 pgs*

OBJECT ORIENTED FORTH, Dick Pountain 242 - $35 1#
Implementation of data structures. First book to make object-oriented programming available to users of even very small home computers. *118 pgs*

SEEING FORTH, Jack Woehr 243 - $25 1#
"... I would like to share a few observations on Forth and computer science. That is the purpose of this monograph. It is offered in the hope that it will broaden slightly the streams of Forth literature ..." *95 pgs*

SCIENTIFIC FORTH, Julian V. Noble 250 - $50 2#
Scientific Forth extends the Forth kernel in the direction of scientific problem solving. It illustrates advanced Forth programming techniques with non-trivial applications: computer algebra, roots of equations, differential equations, function minimization, functional representation of data (FFT, polynomials), linear equations and matrices, numerical integration/Monte Carlo methods, high-speed real and complex floating-point arithmetic. *300 pgs* (Includes disk with programs and several utilities), **IBM**

STACK COMPUTERS, THE NEW WAVE 244 - $62 2#
Philip J. Koopman, Jr. (hardcover only)
Presents an alternative to Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC) by showing the strengths and weaknesses of stack machines (hard-cover only).

STARTING FORTH (2nd ed.), Leo Brodie 245 - $29 2#
In this edition of *Starting Forth*—the most popular and complete introduction to Forth—syntax has been expanded to include the Forth-83 Standard. *346 pgs*

WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++, Norman Smith 270 - $15 1#
This book is about an application language. More specifically, it is about how to write your own custom application language. The book contains the tools necessary to begin the process and a complete sample language implementation. [Guess what language!] Includes disk with complete source. *108 pgs*

ACM - SIGFORTH

The ACM SIGForth Newsletter is published quarterly by the Association of Computing Machinery, Inc. SIGForth's focus is on the development and refinement of concepts, methods, and techniques needed by Forth professionals.

Volume 1 Spring 1989, Summer 1989, #3, #4 911 - $24 2#
F-PC, glossary utility, euroForth, SIGForth '89 Workshop summary (real-time software engineering), Intel 80x8x. Metacompiler in cmForth, Forth exception handler, string case statement for UF/Forth. 1802 simulator, tutorial on multiple threaded vocabularies. Stack frames, duals: an alternative to variables, PocketForth.

Volume 2 #1, #2, #3, #4 912 - $24 2#
ACM SIGForth Industry Survey, abstracts 1990 Rochester conf., RTX-2000. BNF Parser, abstracts 1990 Rochester conf., F-PC Teach. Tethered Forth model, abstracts 1990 SIGForth conf. Target-meta-cross-: an engineer's viewpoint, single-instruction computer.

Volume 3, #1 Summer '91 913a - $6 1#
Co-routines and recursion for tree balancing, convenient number handling.

Volume 3, #2 Fall '91 913b - $6 1#
Postscript Issue, What is Postscript?, Forth in Postscript, Review: PS-Tutor.

1989 SIGForth Workshop Proceedings 931 - $20 1#
Software engineering, multitasking, interrupt-driven systems, object-oriented Forth, error recovery and control, virtual memory support, signal processing. *127 pgs*

1990-91 SIGForth Workshop Proceedings 932 - $20 1#
Teaching computer algebra, stack-based hardware, reconfig-urable processors, real-time operating systems, embedded control, marketing Forth, development systems, in-flight monitoring, multi-processors, neural nets, security control, user interface, algorithms. *134 pgs*

For faster service, fax your orders 510-535-1295

DISKS: Contributions from the Forth Community

The "Contributions from the Forth Community" disk library contains author-submitted donations, generally including source, for a variety of computers & disk formats. Each file is determined by the author as public domain, shareware, or use with some restrictions. This library does not contain "For Sale" applications. *To submit your own contributions, send them to the FIG Publications Committee.*

Prices: Each item below comes on one or more disks, indicated in parentheses after the item number. The price is $6 per disk or $25 for any five disks. 1 to 20 disks = 1 #.

FLOAT4th.BLK V1.4 Robert L. Smith C001 - (1)
Software floating-point for fig-, poly-, 79-Std., 83-Std. Forths. IEEE short 32-bit, four standard functions, square root and log. **IBM.**

Games in Forth C002 - (1)
Misc. games, Go, TETRA, Life... Source. **IBM**

A Forth Spreadsheet, Craig Lindley C003 - (1)
This model spreadsheet first appeared in *Forth Dimensions* VII, 1-2. Those issues contain docs & source. **IBM**

Automatic Structure Charts, Kim Harris C004 - (1)
Tools for analysis of large Forth programs, first presented at FORML conference. Full source; docs incl. in 1985 FORML Proceedings. **IBM**

A Simple Inference Engine, Martin Tracy C005 - (1)
Based on inf. engine in Winston & Horn's book on LISP, takes you from pattern variables to complete unification algorithm, with running commentary on Forth philosophy & style. Incl. source. **IBM**

The Math Box, Nathaniel Grossman C006 - (1)
Routines by foremost math author in Forth. Extended double-precision arithmetic, complete 32-bit fixed-point math, & auto-ranging text. Incl. graphics. Utilities for rapid polynomial evaluation, continued fractions & Monte Carlo factorization. Incl. source & docs. **IBM**

AstroForth & AstroOKO Demos, I.R. Agumirsian C007 - (1)
AstroForth is the 83-Std. Russian version of Forth. Incl. window interface, full-screen editor, dynamic assembler & a great demo. AstroOKO, an astronavigation system in AstroForth, calculates sky position of several objects from different earth positions. Demos only. **IBM**

Forth List Handler, Martin Tracy C008 - (1)
List primitives extend Forth to provide a flexible, high-speed environment for AI. Incl. ELISA and Winston & Horn's micro-LISP as examples. Incl. source & docs. **IBM**

8051 Embedded Forth, William Payne C050 - (4)
8051 ROMmable Forth operating system. 8086-to-8051 target compiler. Incl. source. Docs are in the book *Embedded Controller Forth for the 8051 Family*. **IBM**

68HC11 Collection C060 - (2)
NEW Collection of Forths, Tools and Floating Point routines for the 68HC11 controller. **IBM**

F83 V2.01, Mike Perry & Henry Laxen C100 - (1)
The newest version, ported to a variety of machines. Editor, assembler, decompiler, metacompiler. Source and shadow screens. Manual available separately (items 217 & 235). Base for other F83 applications. **IBM, 83.**

F-PC V3.53, Tom Zimmer C200 - (5)
A full Forth system with pull-down menus, sequential files, editor, forward assembler, metacompiler, floating point. Complete source and help files. Manual for V3.5 available separately (items 350 & 351). Base for other F-PC applications. Req. hard disk. **IBM, 83.**

F-PC TEACH V3.5, Lessons 0–7 Jack Brown C201a - (2)
Forth classroom on disk. First seven lessons on learning Forth, from Jack Brown of B.C. Institute of Technology. **IBM, F-PC.**

VP-Planner Float for F-PC, V1.01 Jack Brown C202 - (1)
Software floating-point engine behind the VP-Planner spreadsheet. 80-bit (temporary-real) routines with transcendental functions, number I/O support, vectors to support numeric co-processor overlay & user NAN checking. **IBM, F-PC.**

F-PC Graphics V4.4, Mark Smiley C203a - (3)
The latest versions of new graphics routines, including CGA, EGA, and VGA suppport, with numerous improvements over earlier versions created or supported by Mark Smiley. **IBM, F-PC.**

PocketForth V1.4, Chris Heilman C300 - (1)
Smallest complete Forth for the Mac. Access to all Mac functions, files, graphics, floating point, macros, create standalone applications and DAs. Based on fig & *Starting Forth*. Incl. source and manual. **MAC**

Yerkes Forth V3.6 C350 - (2)
Complete object-oriented Forth for the Mac. Object access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Incl. source, tutorial, assembler & manual. **MAC,** System 7.01 Compatable.

JLISP V1.0, Nick Didkovsky C401 - (1)
LISP interpreter invoked from Amiga JForth. The nucleus of the interpreter is the result of Martin Tracy's work. Extended to allow the LISP interpreter to link to and execute JForth words. It can communicate with JForth's ODE (Object-Development Environment). **AMIGA,** 83.

Pygmy V1.3, Frank Sergeant C500 - (1)
A lean, fast Forth with full source code. Incl. full-screen editor, assembler and metacompiler. Up to 15 files open at a time. **IBM.**

KForth, Guy Kelly C600 - (3)
A full Forth system with windows, mouse, drawing and modem packages. Incl. source & docs. **IBM, 83.**

ForST, John Redmond C700 - (1)
Forth for the Atari ST. Incl. source & docs. **Atari ST.**

Mops V2.2, Michael Hore C710 - (2)
Close cousin to Yerkes and Neon. Very fast, compiles subroutine-threaded & native code. Object oriented. Uses F-P co-processor if present. Full access to Mac toolbox & system. Supports System 7 (e.g., AppleEvents). Incl. assembler, docs & source. **MAC**

BBL & Abundance, Roedy Green C800 - (4)
BBL public-domain, 32-bit Forth with extensive support of DOS, meticulously optimized for execution speed. Abundance is a public-domain database language written in BBL. Req. hard disk. Incl. source & docs. **IBM HD, hard disk reequired**

WE HAVE CHANGED THE WAY YOU CALCULATE YOUR ORDERS.

1) We have leveled the pricing for FIG items to all members.

2) We have removed the cost of shipping from the price of the items.

3) We have given you a better choice of shipping methods and rates.

Back issues of *Forth Dimensions* and FORML Conference Proceedings are going out of Print!!

For faster service, fax your orders 510-535-1295

fig-FORTH ASSEMBLY LANGUAGE SOURCE

Listings of fig-Forth for specific CPUs and machines with compiler security and variable-length names (see *Installation Manual*, below): - $15 1#

6502	514 - September 80	9900	519 - March 81
6809	516 - June 80	Apple II	521 - August 81
8080	517 - September 79		

fig-FORTH INSTALLATION MANUAL 501 - $15 1#
Glossary model editor—we recommend you purchase this manual when purchasing any of the source code listings above. *61 pgs*

SYSTEMS GUIDE TO fig-FORTH 308 - $25 1#
C. H. Ting (2nd ed., 1989)
How's and why's of the fig-Forth Model by Bill Ragsdale, internal structure of fig-Forth system.

MISCELLANEOUS

T-SHIRT "May the Forth Be With You" 601 - $12 1#
(Specify size: Small, Medium, Large, Extra-Large on order form)
White design on a dark blue shirt.

POSTER (*Oct., 1980 BYTE* cover) 602 - $5 1#

FORTH-83 HANDY REFERENCE CARD 683 - free

FORTH-83 STANDARD 305 - $15 1#
Authoritative description of Forth-83 Standard. For reference, not instruction. *83 pgs*

BIBLIOGRAPHY OF FORTH REFERENCES 340 - $18 2#
(3rd ed., January 1987)
Over 1900 references to Forth articles throughout computer literature. *104pgs*

MORE ON FORTH ENGINES

Volume 10 January 1989 810 - $15 1#
RTX reprints from 1988 Rochester Forth Conference, object-oriented cmForth, lesser Forth engines. *87 pgs*

Volume 11 July 1989 811 - $15 1#
RTX supplement to *Footsteps in an Empty Valley*, SC32, 32-bit Forth engine, RTX interrupts utility. *93 pgs*

Volume 12 April 1990 812 - $15 1#
ShBoom Chip architecture and instructions, Neural Computing Module NCM3232, pigForth, binary radix sort on 80286, 68010, and RTX2000. *87 pgs*

Volume 13 October 1990 813 - $15 1#
PALs of the RTX2000 Mini-BEE, EBForth, AZForth, RTX-2101, 8086 eForth, 8051 eForth. *107 pgs*

Volume 14 814 - $15 1#
RTX Pocket-Scope, eForth for muP20, ShBoom, eForth for CP/M & Z80, XMODEM for eForth. *116 pgs*

Volume 15 815 - $15 1#
Moore: New CAD System for Chip Design, A portrait of the P20; Rible: QS1 Forth Processor, QS2, RISCing it all; P20 eForth Software Simulator/Debugger. *94 pgs*

Volume 16 816 - $15 1#
OK-CAD System, MuP20, eForth System Words, 386 eForth, 80386 Protected Mode Operation, FRP 1600 - 16Bit Real Time Processor. *104 pgs* **NEW**

DR. DOBB'S JOURNAL

Annual Forth issue, includes code for various Forth applications.

Sept. 1982	422 - $5 1#
Sept. 1983	423 - $5 1#
Sept. 1984	424 - $5 1#

FORTH INTEREST GROUP

P.O. BOX 2154 OAKLAND, CALIFORNIA 94621 510-89-FORTH 510-535-1295 (FAX)

Name _____ Phone _____
Company _____ Fax _____
Street _____ eMail _____
City _____
State/Prov. _____ Zip _____
Country _____

U.S. Domestic Postage Rates	Surface	2 day Priority
	$1.00/#	$1.50/#
International Postage Rates	Surface	AIR MAIL
	All /#	1-4 #s/#	>4 #s/#
Canada, Mexico	$1.00	$2.00	$1.30
Other Western Hemisphere	$1.00	$3.25	$2.25
Europe	$1.00	$6.00	$4.50
Other International	$1.00	$8.00	$6.00

Item #	Title	Qty.	Unit Price	Total	#

☐ CHECK ENCLOSED (Payable to: FIG)
☐ VISA ☐ MasterCard
Card Number _____
Signature _____
Expiration Date _____ MEMBERSHIP →

Sub-Total		
10% Member Discount, Member # _____	( )	#s times rate
**Sales Tax on Sub-Total (CA only)		
Postage: Rate _____ x #s		
*Membership in the Forth Interest Group ☐New ☐Renewal $40/46/52		

*MEMBERSHIP IN THE FORTH INTEREST GROUP

The Forth Interest Group (FIG) is a world-wide, non-profit, member-supported organization with over 1,500 members and 40 chapters. Your membership includes a subscription to the bi-monthly magazine *Forth Dimensions*. FIG also offers its members an on-line data base, a large selection of Forth literature and other services. Cost is $40 per year for U.S.A. & Canada surface; $46 Canada air mail; all other countries $52 per year. This fee includes $36/42/48 for *Forth Dimensions*. No sales tax, handling fee, or discount on membership.
When you join, your first issue will arrive in four to six weeks; subsequent issues will be mailed to you every other month as they are published—six issues in all. Your membership entitles you to a 10% discount on publications and functins of FIG. Dues are not deductible as a charitable contribution for U.S. federal income tax purposes, but may be deductible as a business expense.

MAIL ORDERS:
Forth Interest Group
P.O. Box 2154
Oakland, CA 94621
PHONE ORDERS:
510-89-FORTH Credit card orders, customer service.
Hours: Mon–Fri, 9–5 p.m.

PAYMENT MUST ACCOMPANY ALL ORDERS

PRICES: All orders must be prepaid. Prices are subject to change without notice. Credit card orders will be sent and billed at current prices. Checks must be in U.S. dollars, drawn on a U.S. bank. A $10 charge will be added for returned checks.

POSTAGE:
All orders calculate postage as number of #s times selected postage rate. Special handling available on request.

SHIPPING TIME:
Books in stock are shipped within seven days of receipt of the order. Please allow 4–6 weeks for out-of-stock books (deliveries in most cases will be much sooner).

**** CALIFORNIA SALES TAX BY COUNTY:**
7.5%: Sonoma; **7.75%:** Fresno, Imperial, Inyo, Madera, Monterey, Orange, Riverside, Sacramento, San Benito, Santa Barbara, San Bernardino, San Diego, and San Joaquin; **8.25%:** Alameda, Contra Costa, Los Angeles San Mateo, Santa Clara, and Santa Cruz; **8.5%:** San Francisco; **7.25%:** other counties.

For faster service, fax your orders 510-535-1295

RTI1000 Programmable Controller

AUTHOR RECOGNITION PROGRAM

To recognize and reward authors of Forth-related articles, the Forth Interest Group (FIG) adopted the following Author Recognition Program.

Articles

The author of any Forth-related article published in a periodical or in the proceedings of a non-Forth conference is awarded one year's membership in the Forth Interest Group, subject to these conditions:

a. The membership awarded is for the membership year following the one during which the article was published.

b. Only one membership per person is awarded in any year, regardless of the number of articles the person published in that year.

c. The article's length must be one page or more in the magazine in which it appeared.

d. The author must submit the printed article (photocopies are accepted) to the Forth Interest Group, including identification of the magazine and issue in which it appeared, within sixty days of publication. In return, the author will be sent a coupon good for the following year's membership.

e. If the original article was published in a language other than English, the article must be accompanied by an Engish translation or summary.

Letters to the Editor

Letters to the editor are, in effect, short articles, and so deserve recognition. The author of a Forth-related letter to an editor published in any magazine except *Forth Dimensions* is awarded $10 credit toward FIG membership dues, subject to these conditions:

a. The credit applies only to membership dues for the membership year following the one in which the letter was published.

b. The maximum award in any year to one person will not exceed the full cost of the FIG membership dues for the following year.

c. The author must submit to the Forth Interest Group a photocopy of the printed letter, including identification of the magazine and issue in which it appeared, within sixty days of publication. A coupon worth $10 toward the following year's membership will then be sent to the author.

d. If the original letter was published in a language other than English, the letter must be accompanied by an English translation or summary.

```
screen # 64
( IMAGE COMPILER load screen)                ( 7  5 90 bjr 21:53 )
: THRU   1+ SWAP DO  CR I .  .S I LOAD LOOP ;
65 LOAD    ( vocabularies)
68 LOAD    ( image to target)
43 LOAD    ( hex files)      71 LOAD    ( image dump)
72 LOAD    ( multiple dictionaries)
73 LOAD    ( SUPER8 create and compile)
74 75 THRU    ( create, fwd refs)     HOST DEFINITIONS
45 59 THRU    ( SUPER8 assembler)
76 81 THRU    ( Image compiler)
HOST ;S  91 94 THRU    ( test)  HOST ;S
91 112 THRU  ( SUPER8 source code - assembler primitives)
HOST DECIMAL 84 89 THRU    ( SUPER8 source code - high level)
HOST DECIMAL 114 153 THRU
HOST DECIMAL 159 160 THRU  ( initialization values)
HOST ;S

screen # 65
   ( image compiler's vocabularies)          ( 7  6 88 bjr 12:15 )
: AKA   <BUILDS [COMPILE] ' CFA , DOES> @EXECUTE STOP
: IMPORT   IN @ <BUILDS IN ! [COMPILE] ' CFA , DOES> @EXECUTE
   STOP

VOCABULARY HOST IMMEDIATE   HOST DEFINITIONS
AKA NATIVE FORTH IMMEDIATE
AKA EQU CONSTANT

VOCABULARY TARGET IMMEDIATE   TARGET DEFINITIONS
HOST IMPORT HOST  IMMEDIATE   ( must be first defn. in TARGET!)
HOST IMPORT TARGET  IMMEDIATE

HOST DEFINITIONS


screen # 66
( Image to extended memory, byte-swapped) (  8  5 90 bjr  9:20 )
( for 8086 hosts)
CS@ HEX 1000 + CONSTANT TSEG    ( 64K segment for image)
CODE >< ( n - n)    AX POP,    AH AL XCHG,  1PUSH
: T@ ( a - n)    TSEG SWAP @L >< ;
: TC@ ( a - b)    TSEG SWAP C@L ;
: T! ( n a)    SWAP >< TSEG ROT !L ;
: TC! ( b a)    TSEG SWAP C!L ;

: >TCMOVE ( s d n)    BOUNDS DO  DUP C@ I TC!  1+ LOOP DROP ;
: INVOKE ( a)    U. ?COMP ;  ( err msg if exec'ing target word)
DECIMAL


screen # 67
( )
( Image to disk, byte-swapped)

screen # 68
( Image to target machine, byte-swapped)  (  8  5 90 bjr  9:29 )
( for 8086 hosts)
CODE >< ( n - n)    AX POP,    AH AL XCHG,  1PUSH
: T@ ( a - n)    XADR X@+ >< X@+ OR ;
: TC@ ( a - b)    XADR X@+ ;
: T! ( n a)    XADR DUP >< X!+ X!+ ;
: TC! ( b a)    XADR X!+ ;
: >TCMOVE ( s d n)    SWAP XADR  BOUNDS DO  I C@ X!+  LOOP ;

: INVOKE ( pfa)    2+ @ GO AWAIT ;
;S


screen # 69
( )
( Image to extended memory, byte-normal)


screen # 70
( )
( Image to disk, byte-normal)


screen # 71
( Image dump)                               ( 27  5 88 bjr 10:04 )
: (DUMP) \  addr ct ---  | dump as pointed to by reloc
  SPACE  BOUNDS DO  I TC@ 3 .R  LOOP ;
```

AKA defines a synonym word. Usage: AKA newname oldword
IMPORT defines a synonym word of the same name in the current vocabulary. Usage: source-voc IMPORT word

Vocabulary usage for the image compiler:
TARGET holds the "symbol" words for all target definitions. It also holds target compiler directives and target assembler. Within TARGET is a vocabulary tree exactly paralleling the vocabulary tree being built in the image.
HOST is used as an escape to the host's FORTH words.
FORTH is redefined to return to the root target vocabulary...in case it's encountered during the target compilation.

These words store the target image in 8086 extended memory.
TSEG is the segment value for the image. We assume that the 64K following real-forth is available.
>< swaps the hi and lo bytes of the top stack item.

T@ TC@ T! TC@ are the cell and byte, fetch and store operators into the target image.
The image byte order is opposite that of the host.

>TCMOVE copies a string from the host memory to the image.

These words implement Charles Curley's DUMP as part of the image compiler. Use HOST DUMP to look at the image.
Use NATIVE DUMP for the "original" dump of real-Forth memory.

```
: LASCI   \  addr ct ---  | asci type as pointed to by reloc
  SPACE  BOUNDS DO  I TC@  127 AND  DUP
    BL ASCII ~ WITHIN 0= IF DROP ASCII . THEN  EMIT  LOOP ;

: HEAD  \ addr --- | headder for dump display
  16 0 DO I OVER + 15 AND 3 .R LOOP DROP ;

  \ N. B: Not responsible for negative counts! -- the MGT.
: DUMP   \  addr ct ---  | dump as pointed to by reloc
  OVER CR 6 SPACES HEAD  BEGIN  DUP  WHILE  CR OVER 5 U.R
     2DUP 16 MIN >R  R 2DUP (DUMP)  54 TAB LASCI
     R R> MINUS D+  ?TERMINAL IF DROP 0 THEN  REPEAT 2DROP ;
```

```
screen # 72
( Multiple target dictionaries)              (  4  5 90 bjr 15:48 )
( 16 bit addresses)
HOST DEFINITIONS   0 VARIABLE 'DP
: DICTIONARY ( org limit)   <BUILDS SWAP , , DOES> 'DP ! ;
: DP  ( - a)   'DP @ ;
: HERE ( # a)   DP @ ;
: ?FULL   DP 2@ SWAP U< 2 ?ERROR ;     \ error if DP > limit
: ALLOT ( n)   DP +! ?FULL ;
: T, ( n)   HERE T! 2 ALLOT ;
: TC, ( n)   HERE TC! 1 ALLOT ;


0 VARIABLE CONTEXT   0 VARIABLE CURRENT   0 VARIABLE VOC-LINK
: LATEST ( - a)   CURRENT @ T@ ;
( these variables need to be initialized before compilation)
```

These words manage the dictionary being built in the image.
DP HERE ALLOT are analogous to their native forth counterparts, except that they work in 'image addresses'. These words are located in the TARGET vocabulary so they can be found separately from the native forth words in HOST.

T, TC, store words/bytes into the image.

RDP holds the image address of the next available RAM location. Separate DP and RDP are needed when compiling for PROM/RAM. RHERE RALLOT operate on the "ram dictionary".

CONTEXT CURRENT VOC-LINK contain image addresses of the dictionary being built. LATEST returns the image address of the latest definition. (Note the usage: @ T@)
These TARGET words are analogous to their HOST counterparts.

```
screen # 73
( Super8 create and compile)   HEX          (  4  5 90 bjr 14:37 )
( byte-aligned, same name format as host, same width as host)
0 VARIABLE ?HEADS   \ set 0 for headerless
: (TCREATE)   ?HEADS @ IF
      HOST HERE NATIVE LATEST 2DUP  \ dest, src adresses
      C@ 1F AND WIDTH @ MIN 1+ DUP  \ length
      HOST ALLOT  >TCMOVE       \ compile name field in image
      HOST LATEST T,            \ compile link field in image
      HOST CURRENT @ T!  THEN   \ change image vocabulary ptrs
( subroutine threading header)
   ;    \ no header for subroutine threaded code
: TCFA ( a - a)  ;
( subroutine threading compile)
: TCOMP, ( a)   0F6 TC, T, ;    \ a super8 subroutine CALL
: TMARK, ( - a)   0FC TC, HOST HERE 0 T, ;
DECIMAL
```

These words are CPU- and model-specific code.
?HEADS if true, causes headers to be compiled in the image.

(TCREATE) builds a header in the image, linking it into the image vocabulary. The name for the header is obtained from the most recently defined word in the host; thus you must define a host "mirror" word first.
SUPER8 NOTE: no code field is compiled; pfa follows link.
TCFA given a target pfa, returns the target cfa.
TCOMP, compiles a high-level "thread" to a given target adr
Subroutine thread: compile a CALL to the given adr.
TMARK, reserves a high-level "thread", and stacks the target location of the address field for later resolution.
(For forward referencing.)

```
screen # 74
( Change execute and compile actions)     (  1  6 88 bjr 17:44 )
0 VARIABLE 'MIRROR  \ pfa of latest mirror word
: (:)   [ ' : CFA @ ] LITERAL ,  ;

: ACTS:   NATIVE HERE 'MIRROR @ !  (:)
   !CSP NATIVE SMUDGE ]  ;

: ACT   [COMPILE] '  ACTS:  NATIVE COMPILE DROP  CFA ,
   COMPILE ;S SMUDGE [COMPILE] [  ;

\ : MAKES:   NATIVE ] HERE  [COMPILE] DOES>  2+ LATEST PFA CFA !
\    !CSP NATIVE SMUDGE ;

HERE 2+  ] DOES> DUP ( 2+ @ swap) @EXECUTE [
: IMPERATIVE   NATIVE LITERAL 'MIRROR @ CFA !  [ 2 CSP +! ] ;
```

Each target word has associated with it (in the "mirror" word) a "compiling" action and an "executing" action. For most words compiling is "compile my address" and executing is "error".

(:) compiles the cfa for a colon definition in the host.
This is used to make headerless colon definitions.
ACTS: changes the "executing" action of a mirror word.
Usage: ACTS: word word word ;
ACT makes the "executing" action identical to an existing word.
Usage: ACT word
MAKES: changes the "compiling" action of a mirror word.
Usage: MAKES: word word word ;
NOTE that this becomes the executing action as well!
IMPERATIVE makes the "compiling" action of a mirror word the same as its "executing" action. This is akin to IMMEDIATE.

```
screen # 75
( Image create)                           (  8  5 90 bjr  9:21 )
: CREATE   <BUILDS  (TCREATE)  NATIVE HERE 'MIRROR !
   [ ' INVOKE CFA ] LITERAL ,  HOST HERE TCFA ( cfa) NATIVE ,
   DOES> ( a)   STATE @ IF  2+ @ TCOMP, ( compile)
      ELSE  DUP @EXECUTE  ( execute) THEN ; DECIMAL
```

'MIRROR holds the address of the latest-defined mirror word.

CREATE builds a header in the image, and builds a dual-action word in the host dictionary. When executed in compile state, it puts the target word's cfa (Super8: pfa) into the image.
The default action for execute state is an error message.

After CREATE we have enough of the image compiler to compile CODE words (assembler primitives).

```
screen # 76
( Forward references, 16 bit addresses)   (  4  5 90 bjr 15:46 )
HOST DEFINITIONS
: FORWARD   <BUILDS 0 ,  TMARK, ,   [ NATIVE HERE 2+ ]
```

FORWARD builds a root word for a linked list of forward references. When an unknown name is first encountered, FORWARD builds a word by that name with a pointer to where

```
   DOES> ( a)   NATIVE HERE  0 ,  TMARK,
   OVER @ OVER ! SWAP !    [ -2 CSP +! ] ;
CONSTANT (FORWARD)

: RESOLVE ( pfa a)   SWAP BEGIN  2DUP 2+ @ T!  @ -DUP 0= UNTIL
   DROP ;

: CREATE   IN @ >R -FIND R> IN !  CREATE
   IF  DROP DUP CFA @ (FORWARD) = IF  ." ...Resolving"
      HOST HERE TCFA RESOLVE  ELSE DROP THEN  THEN ;


screen # 77
( Image compiling)   HEX                    ( 4  5 90 bjr 14:41 )
HOST DEFINITIONS   0 VARIABLE *LIT*
: TLITERAL ( d)   DPL @ 1+ IF SWAP *LIT* @ TCOMP, T,
   ELSE DROP THEN  *LIT* @ TCOMP, T,  ;

: ?NUMBER ( a - d f)   0 0 ROT DUP 1+ C@ 2D = DUP >R + -1
   BEGIN  DPL ! (NUMBER) 0 OVER C@ ASCII . - UNTIL  DROP ( d a)
   C@ BL = IF  R> IF DMINUS THEN 1  ELSE R> DROP 0 THEN ;

: [   STATE OFF  ( [COMPILE] HOST) ;

: ]   C0 STATE !  BEGIN IN @ -FIND
   IF ( found)  ROT 2DROP CFA EXECUTE
   ELSE  NATIVE HERE ?NUMBER  IF ( number) TLITERAL DROP
      ELSE ( undef)  2DROP  IN ! FORWARD  THEN
   THEN  ?STACK  STATE @ 0= UNTIL ;   DECIMAL

screen # 78
( target interpretation)                ( 13  5 90 bjr 17:18 )
: D>T ( d)   DPL @ 1+ IF SWAP >T ELSE DROP THEN  >T ;

: TINTERPRET   BEGIN -FIND
   IF ( found)  DROP CFA EXECUTE
   ELSE  NATIVE HERE ?NUMBER  0= 0 ?ERROR   ( number) D>T
   THEN  ?STACK AGAIN ;

: TQUIT   BLK OFF STATE OFF  BEGIN
   RP!  CR QUERY TINTERPRET ." Tok" AGAIN ;

: HOT   ' TQUIT CFA 'QUIT ! ."  Tok" QUIT ;
: COOL  ' (QUIT) CFA 'QUIT ! ."  ok" QUIT ;



screen # 79
( Utility words: equ label gap zap seal)  ( 30  5 88 bjr  7:06 )
HOST DEFINITIONS
: SEAL   [COMPILE] ' CFA 2- OFF ;
\ : ZAP    [COMPILE] ' NFA BL TOGGLE ;

\ NATIVE AKA EQU CONSTANT
\ : LABEL   HOST HERE EQU ;
\ : GAP   HOST 2 ALLOT ;            ( word machines)
HEX
: STOP   HOST ?CSP [ ;   IMMEDIATE
: IMMEDIATE   HOST LATEST DUP TC@  40 XOR  SWAP TC! ;
DECIMAL


screen # 80
( Support for defining words)           ( 7  6 88 bjr 20:22 )
0 VARIABLE TODO

: (DOES>)   R> DUP  2+ HOST 'MIRROR @  !   ( host's def'd actn.)
   @  'MIRROR @ 2+ @ 1+ T! ;    ( change image's defined action)

: DOES>   NATIVE COMPILE (DOES>)  TODO @ ,  (:) ;
   NATIVE IMMEDIATE




;
```

its address should be compiled. Subsequent references cause headerless pointers to be linked onto a list. Last link=0. When the word is finally defined, it should be RESOLVEd.
(Note: IN must be restored after -FIND, to use FORWARD.)

RESOLVE name fills the forward reference list starting at pfa with the given value a . (pfa is the pfa of the root word built by FORWARD.)

CREATE is redefined so that, if the word already exists as a forward reference word, it is resolved with the new cfa.

LIT must be filled with the CFA of the LIT primitive, before any colon definitions with literals are attempted.
TLITERAL compiles a single or double literal into the image.

?NUMBER works like NUMBER, except that it returns a flag indicating if the conversion was successful.

[sets interpreting state, and sets CONTEXT to HOST so that host words have precedence in search order.

] sets compiling state, and enters the image compiling loop. Words from the input stream are searched (in the TARGET vocabulary) and executed. The execution action of a defined target word is to compile itself. Other words, such as compiler directives, perform their programmed action.

DOCOL must be filled with the address of the colon CODE, before any colon definitions are made. This is the value which is stuffed into the CFA of all colon defs.
SUPER8 ONLY: no CFAs; the ENTER opcode is stuffed instead.

: sets up for a colon definition in the image, builds the header (with the appropriate CFA), then enters compile mode.

;S must be filled with the address of the ;S primitive, before any image colon definitions are made.

; ends an image colon definition.

After ; we have enough of the image compiler to compile simple colon definitions.

SEAL name makes this word the end of a dictionary chain.
ZAP name removes (smudges) this word from dictionary searches

These are various compile-time directives.
EQU builds a CONSTANT in the TARGET dictionary, but nothing in the image. EQU'd values will not compile, even as literals!!
LABEL EQU's the current compile address in the image.
GAP leaves room in the image for a compiled Forth word.

These words allow the host machine to correctly build "defining" and "defined" words in the target.

TODO holds the ;CODE or DOES> code address just defined in the image.

(DOES>) when executed by the host machine, changes the execute action of the most recently defined target word, in the image AND in the host's mirror word. The image's code address is set to the contents of TODO. The host's "execute" vector is set to the address immediately following the (DOES>) .
DOES> compiles (DOES>) & builds a headerless colon definition in the host for the DOES> action.
Usage: HOST ACTS: word word word DOES> word word word

Refer to the target's source code for DOES> and ;CODE .

```
screen # 81
( Seal target vocabulary)                    ( 7  6 88 bjr 12:26 )
TARGET DEFINITIONS       ( first get a few more needed words)
HOST IMPORT CODE
HOST IMPORT IMMEDIATE
HOST IMPORT ;S
HOST IMPORT (
HOST IMPORT HEX
HOST IMPORT \
HOST IMPORT STOP

TARGET SEAL HOST         ( now seal at the first word in TARGET)

HOST ;S



screen # 82

screen # 83
( Test interactive assembly)                 ( 8  5 90 bjr  9:55 )
HOST HEX C030 FFFF DICTIONARY PROM  PROM  \ origins

CODE LEDOUT   HERE EQU $1   LD R8 # 0FF   LDC 0FFE0 R8
   LD R8 # 0FF   LDC 0FFD0 R8   LD R8 # 0FE   LDC 0FFE0 R8
   LD R2 # 4   BEGIN,   LDC 0FFD0 R0   NOP NOP NOP NOP NOP
      LDC 0FFD0 R1   NOP NOP   DEC R2   Z UNTIL,
  RET  ;C
HERE U.
CODE DEMO   LDW RR0 # 0F000   LDW RR12 # 1234   LDW RR14 # 0F000
   BEGIN,   CALL $1   BEGIN,   DEC R2   Z UNTIL,
      INCW RR0   Z UNTIL,   RET  NOP  NOP  NOP
   ;C

HOST ;S


screen # 84
   ( Super8 : ;)                             ( 7  6 88 bjr 12:26 )
TARGET CODE :   ENTER,  HOST ] TARGET
   ?EXEC !CSP CURRENT @ CONTEXT ! CREATE -2 ALLOT ] ;S
   HOST [
HOST ACTS: ( a)   DROP !CSP  NATIVE CURRENT @ CONTEXT !
   HOST CURRENT @ CONTEXT !  CREATE -2 ALLOT ] ;

TARGET : ;   ?CSP COMPILE ;S SMUDGE [ ;S  HOST [  IMMEDIATE
HOST ACTS: ( a)   DROP ?CSP TARGET ;S HOST [  ; IMPERATIVE

HOST ;S


screen # 85
   ( Super8 dodoes does> (;code)   HEX    ( 7  6 88 bjr 12:30 )
TARGET CODE DODOES ( - a)   TOS 1+ SP @ LDEPD,  TOS SP @ LDEPD,
   TOS POP,  TOS 1+ POP,  NEXT,   \ pop rtn stack to parm stack

TARGET : (;CODE)   R> LATEST PFA 2- ! ;

HOST : ;CODE  HOST ?CSP  TARGET (;CODE)  HOST HERE TODO !
   [ ENTERCODE ;

TARGET : DOES>   COMPILE (;CODE)  1F C,  COMPILE DODOES ;
   IMMEDIATE
HOST ACTS: ( a)   DROP  TARGET (;CODE)  HOST HERE TODO !
   1F TC,  TARGET DODOES  HOST ;

HOST ;S

screen # 86
   ( Super8 constant variable)               ( 12 11 88 bjr 20:03 )
TARGET : CONSTANT   CREATE SMUDGE , ;CODE
   TOS 1+ SP @ LDEPD,  TOS SP @ LDEPD,
   IP W LDW,  W @ TOS LDCI,  W @ TOS 1+ LDC,  EXIT,
HOST ACTS: ( a)   DROP  CREATE T,
   HOST DOES> ( a)   2+ @ 3 + T@ ;

TARGET : VARIABLE ( n)  CONSTANT ;CODE
   TOS 1+ SP @ LDEPD,  TOS SP @ LDEPD,  IP TOS LDW,  EXIT,
HOST ACTS: ( a)   DROP  CREATE T,
   HOST DOES> ( a)   2+ @ 3 + ;

HOST ;S
```

TARGET is the root of the "mirrored" dictionary tree, which will be built in the host. This tree will hold all of the "mirror" words and will exactly duplicate the search order of the dictionary being built in the image.

Once the TARGET vocabulary is sealed, the only exits are
HOST to select the HOST vocabulary
CODE to create a code header and select HOST ASSEMBLER

Note that the vocabulary must be sealed at its first definition, which in this case is the just-defined HOST synonym.

A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

When I started Fast FORTHward, I promised to use it to share essays about Forth, essays about marketing issues, and essays aimed at educating others about Forth. I am pleased to be able to share with you the excerpt concerning threading models from Jack Woehr's essay "Seeing Forth" in his book by the same name. —Mike Elola

Excerpt from "Seeing Forth"

by Jack Woehr

Forth has traditionally a very simple execution engine, but the number [of] Forth implementation strategies can [no] longer be counted on the fingers of one hand. There is perhaps no other computer language whose execution engine exhibits wider and more varied implementations, though Pascal, LISP, BASIC and Prolog are certainly contenders for the crown.

Forth is described as a virtual machine, a software emulation of an imaginary processor which would possess an infinitely extensible instruction set. In the ideal machine, a routine defined in terms of pre-existent operations would become a member of the microprocessor's instruction set.

In order to emulate this ideal processor, the traditional Forth compilers lay down address lists to be stepped through [by the inner interpreter] in the course of executing a Forth word (function). These addresses, for the purpose of the emulation, correspond to the instruction set of the ideal processor.

[...] The hoariest member of the family of Forth inner interpreters is the Indirect-Threaded Interpreter. The body of a colon definition in an indirect-threaded Forth is constructed as follows:

/addr-of-interpreter/ address/ address/ address/...

where

addr-of-interpreter is the address of a routine which will handle the first step of processing the list which follows. Usually the interpreter is a nesting routine, which saves the Instruction Pointer of the caller on the Return Stack and sets the Instruction Pointer to point to the first cell of the following list of addresses.

and

address is the address of a previously-defined Forth word called in the course of executing this definition. The last address in the list of addresses may be the address of an unnesting routine which pops the former Instruction Pointer from the Return Stack.

Forth words executed in this manner continue to nest downwards into lower- and lower-level words until they reach a definition constructed as follows:

/address-of-next-cell/ code/ code/ code/ code/ next/

where

address-of-next-cell is just that, the address of the body of the definition itself. This definition is code and posesses no interpreter which must be pointed to. Simply stepping into itself is sufficient, and it will clean up after itself and begin the process of nesting back upwards as described below.

and

code is executable machine code.

and

next is either a jump to, or the inline expansion of a routine which causes the contents of the cell pointed to by the current Instruction Pointer to be fetched and fed to the interpretive engine, post-incrementing the Instruction Pointer in the process. In other words, this level of Forth execution is the beginning of the end for a Forth Machine Cycle.

Closely related to the Indirect-Threaded Interpreter is the Direct-Threaded Interpreter. The body of a colon definition in a direct-threaded Forth is constructed as follows:

/interpreter-inline/ address/ address/ address/...

where

interpreter-inline is the the actual routine that will handle the first step of processing the list which follows. As above, the interpreter is usually a nesting routine, which saves the Instruction Pointer of the caller on the Return Stack and sets the Instruction Pointer to point to the first cell of the following list of addresses.

and

address is the address of a previously-defined Forth word called in the course of executing this definition. The last address in the list of addresses may be the address of an unnesting routine which pops the former Instruction Pointer from the Return Stack.

Once again, Forth words executed in this manner continue to nest downwards into lower- and lower-level words until they reach a definition constructed as follows:

/code/ code/ code/ code/ next/

where

this definition is code and posesses no interpreter which must be pointed to. Execution commences at the first instruction cell. Stepping into itself is sufficient, and it will clean up after itself and begin the process of nesting back upwards as described below.

and

code is executable machine code.

and

next is either a jump to, or the inline expansion of a routine which causes the contents of the cell pointed to by the current Instruction Pointer to be fetched and fed to the interpretive engine, post-incrementing the Instruction Pointer in the process. In other words, this level of Forth execution is the beginning of the end for a Forth Machine Cycle.

(Continued on page 32.)

Benchmarks Wanted

In late July, the Forth Interest Group (FIG) received a letter from China. The Society of Forth Application Research (SOFAR) there was organizing a large-scale promotion of the Forth language. For Forth vendors and other Forth advocates, here was a golden opportunity to help promote Forth worldwide:

"We are urgently in need of material concerning the comparisons of Forth with languages [such as] C, Pascal, and assembly and other comparisons like arithmetic and general processing. These are needed in the form of performance briefs or testing reports that have source code listings, comparisons of length and speed, etc.

"In addition we would like to know about the fields or businesses which have set Forth as their standard language. [...] We sincerely look forward to your earliest response and assistance on this matter by the 30th of July, 1992. You can contact us through: 10 Third lange, North Street, XiSSi, Beijing, Postal Code 100034, China."

My response to SOFAR has been merely to direct their request to several of the Forth language vendors, asking them to reply directly to SOFAR as well as send me a copy of their response. So far, I have not received anything.

Information such as that requested is of vital importance to support a manager's decision to use Forth. Unfortunately, it can be difficult to find out how Forth measures up.

FIG can act as a channel for information supplied by the vendors—or FIG can generate its own information. Either way, I think FIG needs to be a supplier of such information. I would like for FIG to publish fig-Forth, eFORTH, F83, and F-PC benchmark comparisons with assembly language. With help from the vendors, I would like to see FIG distribute benchmarks of subroutine-threaded, direct-threaded, and indirect-threaded Forths relative to assembly language. FIG should also distribute information regarding the performance improvements possible from optimization techniques. I'll gladly organize the information.

Prospective Forth users may not give Forth its due consideration if we cannot offer information such as this. So if you have any of this information, please mail it to me in care of the FIG office.

Product Watch

JULY-AUGUST 1992

In July, Creative Solutions, Inc. announced a 4.2.2 release of MacForth® Plus (4.2 shipped last January and included MacsBug Interface, editor enhancements, and 68040 compatibility). Upgrades range from $5 to $69 depending on the 4.X version you are upgrading from. As of August, they were still offering a $99 upgrade for the now-defunct Mach2 Forth with proof of ownership. In August, they announced a new Hurdler® card containing a SCSI port as well as four serial ports at a limited-time introductory price of $595.

JULY 1992

The Saelig Company offers the TDS2020 16-bit computer that now accepts up to two TDS2020CM daughter boards with removable SRAM card memory for up to 8Mb of nonvolative memory. It uses the industry standard JEIDA/PCMCIA 68-pin cards. The TDS2020 includes 10-bit A/D, real-time clock, and interfaces for keyboard, LCD, and graphics LCD. A related product is the TMB-200-03 which plugs into a PC to provide a ThinCard drive that accepts the JEIDA/PCMCIA card memories.

AUGUST 1992

Bradley Forthware announced Forthmacs-386, a 32-bit Forth similar to the 680x0 and SPARC workstation versions of the same product. DOS Extender capability is included to provide a full 32-bit environment under DOS, DESQview, and Windows. A ROMable version was also announced.

Forth, Inc. announced a $195 evaluation version of its EXPRESS Event Management and Control System™, a process-control software package. ExpressLite can exercise all EXPRESS functions. For example, the graphics subsystem can be used to create a visual representation of any of your controlled devices in such a way that it is updated to reflect its simulated status. Although I/O drivers are lacking, up to 256 I/O points can be simulated. It also comes with the EXPRESS Technical Manual. The full package sells for $6,875.

Companies Mentioned

Bradley Forthware
P.O. Box 4444
Mountain View, California 94040
Phone: 415-961-1302
Fax: 415-962-0927

Creative Solutions, Inc.
4701 Randolph Road, Suite 12
Rockville, Maryland 20852
Phone: 301-984-0262

Forth, Inc.
111 N. Sepulveda Blvd.
Manhattan Beach,
California 90266-6847
Phone: 310-372-8493
Fax: 310-318-7130

The Saelig Company
1193 Moseley Road
Victor, New York 14564
Phone: 716-425-3753
Fax: 716-425-3835

Character Graphics

C.H. Ting
San Mateo, California

This lesson uses the simplest examples to illustrate the principles of Forth programming: building new instructions from the existing instruction set.

We will use the simple Forth instruction `." xxxx"` to display characters on the screen, and will also use it to build an instruction set which will allow us to construct any block characters on the screen.

To illustrate the use of the `."` instruction, let's write the first Forth program:

```
: hello ." Hello, world!" ;
```

Now, when you type the word *hello* and a return on your keyboard, the characters *Hello, world!* will appear folowing your typed hello.

Explanation:

`:`	Start a new instruction
`hello`	Name of the new instruction
`."`	Print the character string until, but not including, the next "
`;`	Terminate the new instruction

`Hello` is now a new instruction whose function is to print the string *Hello, world!* to the screen. This is the first program most computer courses use to introduce you to a computer language.

Now, what we want to do next is to use this simple technique to display large, block-shaped English alphabets on the screen.

Let's use the letter F as an example:

```
: bar   cr ." *****" ;
: post  cr ." *    " ;
: F     bar post bar post post post ;
```

Type the letter F followed by a carriage return on your keyboard, and you will see a large F character displayed on the screen, like this:

```
*****
*
*****
*
*
*
```

Here we recognize that the character F has two components: a bar composed of five asteriks and a post which can be represented by one or more single asterisks. Therefore, we define two new instructions `bar` and `post` which, respectively, display five asteriks and one asterik. The final instruction `F` can then be defined, which displays a bar, a post, a bar, and then three posts.

The instruction `cr` starts a new line and causes the subsequent characters to be displayed from the left margin of the screen.

Exercise 1: Using the new instructions `bar` and `post`, define new instructions `C`, `E`, and `L` which display the corresponding block characters on the screen.

Exercise 2: Analyze your own surname. Define a set of instructions like `bar` and `post` and use them to construct all the characters in your surname. I will construct my name TING as an example:

```
: center cr ."   *   " ;
: sides  cr ." *    *" ;
: triad1 cr ." * * *" ;
: triad2 cr ." **   *" ;
: triad3 cr ." *   **" ;
: triad4 cr ."  *** " ;
: quart  cr ." ** **" ;
: right  cr ." * ***" ;

: T     bar center center
        center center center center ;

: I     center center center
        center center center center ;

: N     sides triad2 triad2
        triad1 triad3 triad2 sides ;

: G     triad4 sides post
        right triad1 sides triad4 ;

: TING  T I N G ;
```

Exercise 3: It is easy to construct English alphabets this way. The question is, how many primitive instructions are needed to construct all the 26 upper-case letters in this 5 x 7 block format? How about the other characters?

Exercise 4: In principle, we can construct all the Chinese characters using similar techniques. However, most Chinese characters require an enlarged 16 x 16 block format; the more complicated Chinese characters may require a 24 x 24 block. Try to construct a few simple Chinese characters using the 5 x 7 format.

Dr. C.H. Ting is a noted Forth authority who has made many significant contributions to Forth and the Forth Interest Group. His tutorial series will continue in succeeding issues of *Forth Dimensions*.

Styling Forth to Preserve the Expressiveness of C

Mike Elola
San Jose, California

Part of the expressiveness of other programming languages arises from their syntaxes for function calls and expressions. These syntaxes help "package" the flow of function parameters in a way that is easily distinguished.

Most programming languages use one syntax format for function calls, one syntax format for conditionals, and one syntax format for expressions. C is no exception.

The code that is packaged as C expressions always generates a single value. This property of expressions is of key importance. Expressions may be very simple, as exemplified by a variable reference. Or they may be very complex, such as when they use nested expressions. Nevertheless, they are all ultimately reduced to a single value by various binary and unary operations. This packaging lends the programmer an easy "handle" with which to recognize the processing of values and the flow of parameter values into various called routines.

Sometimes parentheses are used to package expressions as part of their incorporation into other units. For example, parentheses appear around expressions that are part of the syntax for branch and loop conditionals. Various syntax formats are thereby combined, yet it is easy to see where one ends and the next begins.

Consider the boost we'd enjoy if Forth compiled C source code...

Besides its simplicity, Forth's freedom from multiple syntax formats—and its freedom from symbols reserved for distinguishing between them—is the source of some confusion regarding where parameter values are being generated and where they are being consumed (see Figure One-a). Stack comments are an attempt to make up for the lack of visual cues (Figure One-b), but they are not always provided.

As you declare a C function, you also declare how references to it will appear as enforced by the compiler: each of its input parameters must be separated by a comma, and no more and no less than the declared number of parameters must be supplied (each of the correct declared type).

However, for most arithmetic operations, an algebraic syntax format is fashionable. In that notation, the generation and passing of parameters lacks the delimiting symbols that are a requirement for the use of functions.

Because we are able to recognize unary and binary arithmetic operations and properly ascertain their input parameters within algebraic notations, many languages do not require us to write code only using a function-oriented syntax. Nevertheless, most languages leave us the ability to create a purely functional syntax. By declaring a function for addition, for example, we can write the following code: `add(1,1)`.

Switching to a functional syntax may be considered a partial step towards (Forth) postfix notation. Forth has taken a bigger step towards a uniform syntax by abandoning support for algebraic notation. Nevertheless, Forth hangs on to the symbols of algebraic notation as the names of its functions. As long as most languages continue to define those symbols as infix arithmetic operators, they cannot allow you to redefine those symbols as the names of functions. Generally, you cannot expect to use code such as: `+(1,1)`. Forth offers more freedom in the names you assign to functions due to its relative lack of reserved meanings for symbols.

C shows a slight movement in the direction of syntax consolidation, particularly if you look at its repetition constructs that have been packaged as functions, such as `while()` and `for()` loops. For its conditional statements, however, C still resorts to an alternate format involving open and close braces around blocks of code. Forth does a more thorough job of integrating its language elements into a uniform syntax format.

Regularization steps such as these are what have led to Forth's simplicity and compactness: it abandons support for several syntax formats, streamlining its parsing requirements. While most of the accompanying effects are good ones, there might have been undesired consequences. We may be overlooking how a simpler parsing model has impaired the expressiveness of Forth source code.

Taken together, these two measures afford levels of expressiveness that Forth cannot equal: (1) the use of parentheses for subexpressions that generate values; and (2) the use of parentheses and commas to distinguish the end, the beginning, and the continuation of input parameters for a function. Statements such as

```
printf("The value is: %i", int(sqrt(3))
```

convey clearly how many parameters are passed to each function and what happens with the values returned by each of the functions. Furthermore, the notation is very compact.

How clear is it that PRINTF in Figure One-a requires two stack parameters? The misleading visual cues in Figure One-a suggest two unary functions, one (SQUARED) that takes a number as its input and another (PRINTF) that takes a string as its input. Forth code needs to make clear how many parameters are being passed to each routine. Stack comments are the usual way we go about this, as shown in Figure One-b.

Figure One-a.

```
3 SQUARED
"The value is: %i" PRINTF
```

Figure One-b.

```
3 SQUARED
"The value is: %i"
( product addr -- ) PRINTF
```

The coexistence of several syntaxes in languages such as C contributes to the easy visual subdivision of source code, improving its readability. You can easily subdivide such code into spans that correspond to the generation of values and spans that correspond to the consumption of values, with reserved symbols punctuating the various transitions. Since many programmers have strong math backgrounds, they learn this notation quickly and view it in a friendly way.

So expressiveness is largely a matter of packaging. Furthermore, Forth's syntax fails to package code so that the flow of parameters is unmistakable.

These concerns prompted me to take up the challenge of designing a new Forth styling convention.

Styling Forth for Parameter Flow

Our indentation of Forth code provides important cues about the start and end of a control-flow construct. I propose that we also use indents to provide visual cues about the start and end of a block of code that generates the parameters for a Forth routine. (I spent considerable time trying to coerce other symbols to serve the same purpose, but I had no success.)

The startling—or perhaps amusing—part of this proposed indenting convention is that it is a postfix convention, since input parameters always precede the Forth word that uses them. Furthermore, any code that generates parameters is placed on its own line to help distinguish parameter generation as well as in a C function call—where commas serve a similar purpose. The result is postfix indents that are part of a vertically oriented specification:

```
    3
  SQUARED
  "The value is: %i"
PRINTF
```

To make the format of the code less vertical and somewhat more compact, consider placing any unary operation on the same line as the code that generates its input—but still allow a separate line for the duo:

```
3 SQUARED
"The value is: %i"
PRINTF
```

This styling convention looks its silliest when we write simple arithmetic expressions:

```
    3
    4
  *
  2
+
```

While I don't expect these conventions to win immediate favor, they could help someone learning Forth. If a uniform syntax is a Forth virtue, then a uniform indenting convention can also be a virtue, despite its occasional spaciousness.

A Pretty-Printer Challenge

Rather than enter code according to these style guidelines, we could develop a pretty printer to create the indentations. (This is left as an exercise for the reader, as usual.)

Such a tool would help make all prior Forth code more expressive, regardless of the originator's reluctance to include stack comments. Further, such a tool will suggest how we might create source-code checkers that can detect stack errors without debugging effort.

To make the pretty printer even more challenging, consider that Forth source code typically contains stack-manipulating words that introduce artificial separation between input parameters and the routines that use them. For example, try adding parameter-flow indentations to the following code:

```
4
6
SWAP
8
*
+
```

One solution might be to introduce comments to show the values that were unprocessed, yet were specified in positions that made them appear as if they would be processed:

```
          4
          6
        SWAP (
      4 )
      8
    * (
  x
  6 )
+
```

A Way to Eliminate Forth's Stack Operators

Because every expression and every function in C generates one and only one value, I anticipate that the conversion of C programs to Forth will never require Forth's stack operators.

The only occasion when a value may be generated in the wrong position on the stack is when an expression or function is able to generate more than one value (which they cannot do in languages such as C). With the extra flexibility

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!
- Faster than optimized C!
- Compile 40,000 lines per minute! (10 Mhz 286)
- 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!
- Source code for 2500 functions!
- Data structures, control structures and interface protocols from any other language!
- Implement borrowed features, more efficiently than in the source!
- An architecture that supports small programs or full megabyte ones with a single version!
- No byzantine syntax requirements!
- Outperform the best programmers stuck using conventional languages! (But only until they also switch.)

HS/FORTH with FOOPS - The only full multiple inheritance interactive 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
```

The painful OOL learning curve disappears when you don't have to force the world into a hierarchy.

WAKE UP ! ! !

Forth need not be 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 that could otherwise be making your life easier now! 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 and often better than in the parent language.

Speaking of assembler language, learning it in a supportive Forth environment virtually eliminates the learning curve. People who failed previous attempts to use assembler language, often conquer it in a few hours using HS/FORTH. And that includes people with NO previous computer experience!

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. Source code is in ordinary ASCII text files.

HS/FORTH supports megabyte and larger programs & data, and runs as fast as 64k limited Forths, even without automatic optimization -- which accelerates to near assembler 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.

PERSONAL LEVEL $299.
NEW! Fast direct to video memory text & scaled/clipped/windowed graphics in bit blit windows, mono, cga, ega, vga, all ellipsoids, splines, bezier curves, arcs, turtles; lightning fast pattern drawing even with irregular boundaries; powerful parsing, formatting, file and device I/O; DOS shells; interrupt handlers; call high level Forth from interrupts; single step trace, decompiler; music; compile 40,000 lines per minute, stacks; file search paths; format to strings. 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 delivers machine code speed.**

PROFESSIONAL LEVEL $399.
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 $499.
Metacompiler: DOS/ROM/direct/indirect; threaded systems start at 200 bytes, Forth cores from 2 kbytes; C data structures & struct+ compiler; MetaGraphics TurboWindow-C library, 200 graphic/window functions, PostScript style line attributes & fonts, viewports.

ONLINE GLOSSARY $ 45.

PROFESSIONAL and PRODUCTION LEVEL EXTENSIONS:

FOOPS+ with multiple inheritance **$ 79.**
TOOLS & TOYS DISK $ 79.
286FORTH or **386FORTH $299.**
16 Megabyte physical address space or gigabyte virtual for programs and data; DOS & BIOS fully and freely available; 32 bit address/operand range with 386.
ROMULUS HS/FORTH from ROM **$ 99.**

Shipping/system: US: $9. Canada: $21. foreign: $49. We accept MC, VISA, & AmEx

of one routine generating multiple outputs comes the possibility that other routines may require the same parameters to be supplied in an alternate sequence.

This suggests that one way to rid Forth of its stack operators is to exclusively define words that generate no more than one parameter. This imparts to Forth the notational granularity needed to give back control of stack configuration (or parameter flow) through notational means: you merely order your code to reflect how you want the stacked parameters ordered.

Benefits of the Verbosity Requirements of C

In C, each item in the input parameter list of a function is filled by an expression—and an expression must always produce exactly one value. This correspondence helps generate dynamic syntax requirements for each function call that must be satisfied: you must always call the function in a consistent manner by specifying an expression for each of its declared parameters.

In C, the flow of parameters cannot be factored across two routines—the way they can in Forth—at least not notationally. In Forth, we are free to compile definitions where there is no mention of missing inputs. Even though MOD requires two parameters, we are free to compile the following definition of MOD6, in which the missing parameter for MOD becomes an input requirement for the declared routine, MOD6:

```
: MOD6 6 MOD ;
```

The same level of factoring granularity is available in C, but the code must be written more verbosely: you must explicitly specify all of the input parameters that flow into each called routine. Notationally and otherwise, there can be no mistaking the fact that a modulus operation requires two parameters. So C requires the explicit specification of both inputs for the modulus operation, aided by a placeholder that represents the input parameter supplied to MOD6.

```
    /* code that once compiled can    */
    /* be linked into numerous appli- */
    /* cations without redefinition.  */
MOD6(input)
int input;
{
  return(input % 6);
}
```

Comparatively, the Forth notation is abbreviated. This helps afford Forth its macro-assembler feel.

I fear that this short-cut has also inhibited the development of Forth libraries. I feel that a Forth library mechanism should be created that can faithfully match the features of C libraries.

Looking Forward

Whereas the evolution of the many other programming languages tends to reveal an incremental refinement of earlier ones, Forth seems to be a major departure from its peer languages. To help demystify Forth to others, we need to note its similarities to other languages as often as we can. The indentation styling I have suggested adds visual cues to Forth code such that it becomes much easier to correlate Forth code to that of other languages.

Another helpful exercise is to try to translate C source code into Forth. Such an exercise should make clear further similarities and differences between these two languages.

This article could be considered an introductory one in a series focused on the issues of translating C to Forth. I do not feel adequately qualified for that undertaking. Perhaps it can take the form of an article contest.

Consider the boost we might enjoy if Forth supported the compilation of C source code as a readily available option. For example, we could decisively lay to rest the old argument that there are too few Forth programmers to support Forth.

(Fast Forthward, continued from page 27.)

There are a variety of means whereby direct threading is implemented. On a typical Complicated Instruction Set (CISC) processor, all interpreters are carefully designed to be compact and speedy, since they are compiled inline every time the Forth system lays down a colon definition in the dictionary.

The Zilog Z880 (Super 8) took another approach, dedicating a set of registers to the emulation of the Forth virtual machine and providing the one-byte opcodes ENTER ("nest"), EXIT ("unnest") and NEXT in microcode on the processor itself.

Another form of threading commonly used on advanced microprocessors such as the 68000 is Subroutine Threading. A subroutine-threaded Forth possesses colon definition bodies which are pure assembly code. The typical call-by-address scheme of Forth compilation is implemented in machine-code subroutine calls to the CFAs of called definitions. As the entire body of every definition, code or colon, compiles down to code, there is much latitude for a smart compiler to optimize by inline expansion of short definitions instead of call compilation.

There are also Token-Threaded Forths, where addresses refer to entries in a jump table which contains the actual hard adresses where the code resides.

And finally, there are the "Silicon-Threaded" Forths, where the instruction set of the processor and its call mechanism are designed to suit the peculiarities of Forth. The Novix, the Harris RTX Series, and the Silicon Composers SC32 all implement decoding logic which decides if an instruction is an address call or a machine instruction depending on the state of certain bits in the instruction. The result of this scheme is the fastest execution speeds obtainable for threaded code.

The arbitrary categorization performed in the above paragraphs is by no means exhaustive. What are we to call, for example, William "Mitch" Bradley's CForth83, a Forth system primarily aimed at *NIX systems, in which the user dictionary is JSR-Threaded but the kernel is a gigantic C-language "switch" statement?

Backburner code. 8051 assembler (see article, pages 38–39).

```
index
600 8051 assembler/primary load block
601 8051 assembler/secondary load block
602 support
603 virtual array support
604 revectored  NUMBER
605 operand definition
606 operand vectoring
607 operand vectoring
608 vector definition
609 vector definition
610 destination vectoring
611 instruction definition
612 instruction classes
613 instructions
614 instructions
615 instructions
616 instructions
617 vectors
618 vectors
619 vectors
620 vectors
621 vectors
622 vectors
623 vectors
624 vectors
625 vectors
626 vectors

block 600
 0 ( 920910/8051 assembler/primary load block)
 1 ( support) 602 603 THRU
 2 ( application) 601 LOAD
 3
 4
 5 ( initialization) ' [NUMBER] 'NUMBER !
 6
 7 EXIT
 8
 9 current memory requirement above FORTH nucleus & electives:
10    13 547 bytes
11
12
13
14
15

block 601
 0 ( 920910/8051 assembler/secondary load block)
 1 ( operand definition) 605 LOAD
 2 ( redefined  NUMBER ) 604 LOAD
 3 ( operand vectoring) 606 607 THRU
 4 ( vector definition) 608 609 THRU
 5 ( destination vectoring) 610 LOAD
 6 ( instruction definition) 611 LOAD
 7 ( instruction classes) 612 LOAD
 8 ASM DEFINITIONS
 9 ( instructions) 613 616 THRU
10 FORTH DEFINITIONS
11 ( vectors) 617 626 THRU
12
13
14
15

block 602
 0 ( 920910/support)  HEX
 1 : FORGET   ['] (NUMBER) 'NUMBER !  FORGET ;
 2 : EMPTY    ['] (NUMBER) 'NUMBER !  EMPTY ;
 3
 4 001B VOCABULARY ASM
 5
 6 : CCONSTANT ( c)   CREATE C,  DOES> ( - c)   C@ ;
 7
 8 VARIABLE <BASE>
 9 : .S   BASE @ <BASE> !  HEX  .S  <BASE> @ BASE ! ;
10
11
12
13
14
15

block 902
 0 FORGET  is redefined to accommodate the revectoring of  NUMBER
 1 EMPTY  is redefined to accommodate the revectoring of  NUMBER
 2
 3 vocabularies already defined or reserved in 8086 polyFORTH:
 4   normal use:       0001 FORTH   0013 ASSEMBLER   0015 EDITOR
 5   metacompilation:  0071 FORTH   0017 HOST        0179 ASSEMBLER
 6 8051ASM  specifies a vocabulary linked to the vocabulary FORTH
 7
 8 CCONSTANT  provides a byte-size constant
 9
10 <BASE>  preserves the value of  BASE  during stack display
11 .S  is redefined to display the stack in hexadecimal and then
12   return to the previous numeric base
13
14
15
```

Advertisers Index

```
block 603
 0 ( 920910/virtual array support)
 1 2400 CONSTANT VARRAY
 2 : VH ( - a)    VARRAY BLOCK ;
 3 : VHERE ( - n)    VH @ ;
 4 : VADDRESS ( elem - a)    1024 /MOD VARRAY + BLOCK + 2+ ;
 5 : VC! ( c elem)    VADDRESS C!  UPDATE ;
 6 : VC@ ( elem - c)    VADDRESS C@ ;
 7 : VC, ( c)    VHERE VC! UPDATE  1 VH +! UPDATE ;
 8 : VSTORE ( n)    0 DO  VC,  LOOP ;
 9
10 : VDUMP    BASE @ <BASE> !  HEX  VHERE ?DUP IF
11    0 DO  I VC@ U.  LOOP  THEN  <BASE> @ BASE ! ;
12 : VFORGET    0 VH !  UPDATE ;
13
14 ( initialization) VFORGET
15

block 604
 0 ( 920910/revectored  NUMBER )
 1 : [NUMBER]    (NUMBER)  1 SEQUENCE ! ;
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13

block 605
 0 ( 920910/operand definition)  HEX
 1 CVARIABLE CLASS
 2 VARIABLE SEQUENCE
 3 : ?FIRST  ( - n)    SEQUENCE DUP @  1 ROT ! ;
 4 CREATE OPERANDS  2 ALLOT
 5 : :O ( c)    CCONSTANT
 6    DOES>    C@  ?FIRST OPERANDS +  C! ;
 7 ASM DEFINITIONS
 8  1 :O #          2 :O A          3 :O C
 9  4 :O @R0        5 :O @R1
10  6 :O R0         7 :O R1         8 :O R2         9 :O R3
11 0A :O R4        0B :O R5        0C :O R6        0D :O R7
12 0E :O @A        0F :O AB
13 10 :O DPTR      11 :O @A+DPTR  12 :O @A+PC     13 :O @DPTR
14 ( 14 :O /)
15 FORTH DEFINITIONS

block 606
 0 ( 920910/operand vectoring)
 1 11 CONSTANT #CLASSES    ( z-dimension)
 2 20 CONSTANT #OPERANDS
 3 #OPERANDS CONSTANT #X ( x-dimension)
 4 #OPERANDS CONSTANT #Y ( y-dimension)
 5
 6 #OPERANDS #OPERANDS * CONSTANT #XY
 7 #OPERANDS #OPERANDS * #CLASSES * CONSTANT #ELEMENTS
 8
 9 CREATE VECTORS ( 3-dimensional array)  #ELEMENTS 2* ALLOT
10
11 : ELEMENT ( x y z - elem#)    #XY * SWAP #X * + + ;
12 : >VECTOR ( elem# - a)    2* VECTORS + ;
13
14 : !VECTOR ( a elem#)    >VECTOR ! ;
15 : @VECTOR ( elem# - a)    >VECTOR @ ;

block 607
 0 ( 920910/operand vectoring)
 1 : NULL ;
 2 : NULL>VECTORS    #ELEMENTS 0 DO  ['] NULL I !VECTOR  LOOP ;
 3
 4 ( initialization) NULL>VECTORS
 5
 6 : .ALL    CR  #ELEMENTS 0 DO  I @VECTOR 10 U.R  LOOP ;
 7 : .CLASS ( c1)    #CLASSES 1- MIN  CR  #Y 0 DO    #X 0 DO
 8    DUP I J ROT ELEMENT @VECTOR  10 U.R  LOOP LOOP  DROP ;
 9
10 : .OPERANDS    BASE @ <BASE> !  HEX
11     OPERANDS DUP C@ U.  1+ C@ U.  <BASE> @ BASE ! ;
12
13
14
15

block 903
 0 VARRAY  names the first block of the virtual array residing in
 1   an MS-DOS file at the specified offset in the FORTH block map
 2 VH  obtains the block buffer address of the array pointer
 3 VHERE  returns the array pointer, kept in the first cell of the
 4   array; the pointer is the number of the next available byte
 5 VADDRESS  takes an element number and returns the corresponding
 6    block buffer address; the  2+  skips over the array pointer
 7 VC!  stores a byte at the specified byte offset in the array
 8 VC@  fetches a byte from the specified byte offset in the array
 9 VC,  stores a byte into the next available position in the array
10   and advances the array pointer
11 VSTORE  stores the specified number of bytes from the stack into
12   the array
13 VDUMP  displays the array, up to the current value of the array
14   pointer; the array is displayed in hexadecimal
15 VFORGET  resets the array pointer

block 904
 0 [NUMBER]  is a modification of the vectored routine  (NUMBER) ;
 1   encounter of a data or address byte sets  SEQUENCE  to 1, but
 2   does not alter either byte of  OPERANDS ; this modification
 3   allows the assembler to discriminate between instructions of
 4   the form  (data)(operand)(mnemonic)  and those of the form
 5   (operand)(data)(mnemonic) ; otherwise, the single operand
 6   would always leave its value in the first byte of  OPERANDS ;
 7   CAUTION:  before  EMPTYing the dictionary or  FORGETting the
 8   application,  NUMBER  must be  revectored to  (NUMBER)  or the
 9    system will crash, since forgetting the application will also
10   forget  [NUMBER] ; thus, for safety,  EMPTY  and  FORGET  have
11   been redefined to accomplish this; to speed loading, the
12   initial revectoring to  [NUMBER]  is done after the
13   application has been loaded

block 905
 0 execution of an instruction loads the class into  CLASS
 1 SEQUENCE  is used in logging the order of operand encounters; it
 2   is zeroed before assembly of each instruction; see  PREPARE
 3 ?FIRST  obtains from  SEQUENCE  the value 0 when executed by the
 4   first operand, then stores in  SEQUENCE  the value 1, which
 5   is returned when  ?FIRST  is executed by the second operand
 6 OPERANDS  is a byte array which holds, in order of encounter,
 7   the operand numbers of the operands, if any, which apply to
 8   the instruction being assembled; it must be cleared before
 9   assembly of each instruction; see  PREPARE
10 :O  is a defining word for operands; associated with each
11   operand is a constant, the operand number; the value zero is
12   reserved for the operand  NULL ; at run time, the operand
13   stores its number into the appropriate byte of  >VECTORS
14 the operands, defined with  :O , are compiled into the
15   vocabulary  ASM

block 906
 0 #CLASSES  holds the number of potential instruction classes
 1 #OPERANDS  holds the number of potential operands, including
 2   NULL , which corresponds to instructions with no operands
 3 #X , #Y , #XY , &  #ELEMENTS  are named to clarify index
 4   calculations for accessing the 3-dimensional array
 5 VECTORS  is a 3-dimensional array which associates a vector with
 6   each combination of 1st operand, 2nd operand, & instruction
 7   class
 8 ELEMENT  computes the linear element number from the operand
 9   numbers and the instruction class
10 >VECTOR  returns a pointer to the specified element of  VECTORS
11 !VECTOR  stores a pfa into the specified element of  VECTORS
12 @VECTOR  fetches a pfa from the specified element of  VECTORS
13
14
15

block 907
 0 NULL>VECTORS  makes NULL the default vector; executing it before
 1  loading operand behaviours allows one to load only vectors
 2  corresponding to valid combinations of operand pairs and class;
 3  invalid pairings for a particular instruction will not always
 4  execute  NULL , since, in general, not all operand behaviours
 5  defined for a particular class are valid all for instructions
 6  in the class; if classes are limited to a single instruction,
 7  all invalid pairings will be trapped, in which case  NULL  may
 8  be replaced with the word  : INVALID  ." invalid operand(s)" ;
 9 .ALL  displays in linear sequence [ (x0,y0,z0), (x1,x0,z0), ...,
10   (xn,y0,z0), (x0,y1,z0), (x1,y1,z0), ... , (xn,yn,zn) ] all
11   elements of  VECTORS
12 .CLASS  displays in linear sequence all elements of  VECTORS
13   corresponding to the specified instruction class; since the
14   class typically is input manually, it is checked for validity
15 .OPERANDS  displays  OPERANDS , for diagnostic purposes
```

```
block 608
 0 ( 920910/vector definition)
 1 : :V ( op1 op2 cl)    :  LAST @ @ CFA 2+
 2    ROT ROT 2SWAP SWAP 2SWAP ELEMENT !VECTOR ;
 3
 4
 5
 6 EXIT
 7            pfa cl op2 op1
 8             a   z   y   x    need    z   y   x   a
 9 -------------------------
10 ROT         y   a   z   x
11 ROT         z   y   a   x
12 2SWAP       a   x   z   y
13 SWAP        x   a   z   y
14 2SWAP       z   y   x   a

block 609
 0 ( 920910/vector definition)  HEX
 1  1 CCONSTANT =#
 2  2 CCONSTANT =A          3 CCONSTANT =C
 3
 4  4 ~ CCONSTANT =@R0      5 ~ CCONSTANT =@R1
 5
 6  6 CCONSTANT =R0         7 CCONSTANT =R1
 7  8 CCONSTANT =R2         9 CCONSTANT =R3
 8 0A CCONSTANT =R4        0B CCONSTANT =R5
 9 0C CCONSTANT =R6        0D CCONSTANT =R7
10
11 0E CCONSTANT =@A        0F CCONSTANT =AB
12 10 CCONSTANT =DPTR      11 CCONSTANT =@A+DPTR
13 12 CCONSTANT =@A+PC     13 CCONSTANT =@DPTR
14 ( 14 CCONSTANT =/)

block 610
 0 ( 920910/destination vectoring)
 1 VARIABLE (MODE)
 2
 3 : STORE ( n)    (MODE) @EXECUTE ;
 4
 5 : DISPLAY  .S  CR  ABORT ;
 6
 7 : >DISK   ['] VSTORE (MODE) !  VFORGET ;
 8 : >DISPLAY   ['] DISPLAY (MODE) ! ;
 9
10 ( default) >DISPLAY
11
12
13
14
15

block 611
 0 ( 920910/instruction definition)
 1 : PREPARE   0 SEQUENCE !  0 OPERANDS ! ;
 2
 3 ( initialization) PREPARE
 4
 5 : ASSEMBLE ( opc)   OPERANDS DUP C@ ( x)  SWAP 1+ C@ ( y)
 6    CLASS C@ ( z)  ELEMENT >VECTOR @EXECUTE PREPARE STORE ;
 7
 8 : :INSTRUCTION ( opc cl)   CCONSTANT  C,
 9    DOES>   DUP  C@ CLASS C!  1+ C@  ASSEMBLE ;
10
11 : :CLASS ( cl)   CCONSTANT  DOES>   C@  :INSTRUCTION ;
12
13
14
15

block 612
 0 ( 920910/instruction classes)  HEX
 1  0 :CLASS  0CLASS
 2  1 :CLASS  1CLASS
 3  2 :CLASS  2CLASS
 4  3 :CLASS  3CLASS
 5  4 :CLASS  4CLASS
 6  5 :CLASS  5CLASS
 7  6 :CLASS  6CLASS
 8  7 :CLASS  7CLASS
 9  8 :CLASS  8CLASS
10  9 :CLASS  9CLASS
11 0A :CLASS  ACLASS
12 0B :CLASS  BCLASS
13
14
15

block 908
 0 :V  is a defining word for operand/class vectors; it expects on
 1   the stack the operand numbers of the first and second
 2   operands, respectively, followed by the instruction class; it
 3   creates a colon definition and loads into the appropriate
 4   element of  VECTORS  the pfa of that colon definition
 5
 6
 7
 8
 9
10
11
12
13
14

block 909
 0 these byte constants facilitate the definition of vectors
 1
 2 ~  causes the full name to be compiled; polyFORTH normally
 3   compiles only the first 3 characters of the name and the
 4   length, so that the names  =@R0  and  =@R1  would be
 5   indistinguishable
 6
 7
 8
 9
10
11
12
13
14

block 910
 0 (MODE)  holds the pfa of the compilation word ( VSTORE  or
 1    or  DISPLAY )
 2 STORE  is vectored to the previously-selected compilation word
 3   ( VC,  or  DISPLAY ); the argument passed on the stack is the
 4   number of bytes to be compiled; the argument is used by  VC,
 5   and is discarded by  DISPLAY
 6
 7 DISPLAY  displays and then clears the stack; note that assembled
 8   instructions, prior to compilation, are in the form of one or
 9   more bytes on the stack, in proper order for compilation via
10   VC,
11
12 >DISK  vectors  STORE  to  VSTORE , so that assembled
13   instructions are compiled to the virtual disk array  VARRAY
14 >DISPLAY  vectors  STORE  to  DISPLAY ; no code is compiled
15

block 911
 0 PREPARE  is executed before assembly of each instruction; it is
 1   executed at load time to prepare for the first instruction
 2 ASSEMBLE  uses the operand and class numbers to index into the
 3   array  VECTORS , from which it obtains the pfa of a routine
 4   corresponding to the specific operands & sequence of encounter
 5   for the instruction being assembled; the opcode basis is left
 6   on the stack by execution of the instruction; assembled code
 7   is compiled or displayed by the vectored routine  STORE ;
 8   PREPARE  precedes  STORE  to allow use of  ABORT  in  DISPLAY
 9 :INSTRUCTION  defines instructions; it is executed by the
10   run-time behaviour of  :CLASS ; the run-time behaviour of an
11   instruction is to load  CLASS , leave on the stack the basis
12   for the opcode, then invoke  ASSEMBLE
13 :CLASS  defines instruction classes; each instruction class is a
14   defining word for instructions of that class; the run-time
15   behaviour of  :CLASS  executes  :INSTRUCTION

block 912
 0 0CLASS , etc  are instruction classes; each class is a defining
 1   word for instructions of that class
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
block 613
 0 ( 920910/instructions)  HEX
 1  00 0CLASS NOP
 2  03 0CLASS RR
 3  13 0CLASS RRC
 4  22 0CLASS RET
 5  23 0CLASS RL
 6  32 0CLASS RETI
 7  33 0CLASS RLC
 8  73 0CLASS JMP
 9  84 0CLASS DIV
10 0A4 0CLASS MUL
11 0C4 0CLASS SWAP
12 0D4 0CLASS DA
13  93 ~ 0CLASS MOVC
14 0E0 ~ 0CLASS MOVX

block 614
 0 ( 920910/instructions)  HEX
 1  70 1CLASS MOV
 2
 3  20 2CLASS ADD
 4  30 2CLASS ADDC
 5  40 2CLASS ORL
 6  50 2CLASS ANL
 7  60 2CLASS XRL
 8  90 2CLASS SUBB
 9 0C0 2CLASS XCH
10 0D0 2CLASS XCHD
11
12   0 3CLASS INC
13  10 3CLASS DEC


block 615
 0 ( 920910/instructions)  HEX
 1 0D0 4CLASS DJNZ
 2
 3 0B0 5CLASS CJNE
 4
 5 0B0 6CLASS CPL
 6 0C0 6CLASS CLR
 7 0D0 6CLASS SETB
 8
 9  10 7CLASS JBC
10  20 7CLASS JB
11  30 7CLASS JNB
12
13


block 616
 0 ( 920910/instructions)  HEX
 1  40 8CLASS JC
 2  50 8CLASS JNC
 3  60 8CLASS JZ
 4  70 8CLASS JNZ
 5  80 8CLASS SJMP
 6 0C0 8CLASS PUSH
 7 0D0 8CLASS POP
 8
 9   2 9CLASS LJMP
10  12 9CLASS LCALL
11
12 EXIT
13   1 ACLASS AJMP
14  11 ACLASS ACALL
15

block 617
 0 ( 920910/vectors)  HEX
 1  0         0        0 ~ :V V0.00        1 ;
 2 =A         0        0 ~ :V V0.01        1 ;
 3 =DPTR      0        0 ~ :V V0.02        1 ;
 4 =AB        0        0 ~ :V V0.03        1 ;
 5 =@A+DPTR   0        0 ~ :V V0.04        1 ;
 6 =A        =@A+DPTR  0 ~ :V V0.05        1 ;
 7 =A        =@A+PC    0 ~ :V V0.06   10 - 1 ;
 8 =A        =@DPTR    0 ~ :V V0.07        1 ;
 9 =A        =@R0      0 ~ :V V0.08    2+  1 ;
10 =A        =@R1      0 ~ :V V0.09    3 + 1 ;
11 =@DPTR    =A        0 ~ :V V0.10   10 + 1 ;
12 =@R0      =A        0 ~ :V V0.11   12 + 1 ;
13 =@R1      =A        0 ~ :V V0.12   13 + 1 ;
14
15
```

```
block 913
 0 these are instruction mnemonics defined as class 0 instructions;
 1   the number preceding the instruction defining word  0CLASS
 2   is the basis for assembly of the opcode for the instruction
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
```



```
block 917
 0 a behaviour is defined for each possible combination of class
 1   and operand pair; invalid combinations are assigned the null
 2   behaviour; the behaviours are arbitrarily named ( Vz.xx for
 3   "vector xx of class z"), the name is not used, except
 4   for diagnostics;  :(0)  expects on the stack the following
 5   order of operand and class numbers:
 6   operand1# operand2# class#
 7 the number left on the stack by the vector is the number of
 8   bytes (opcode + data/address) to be compiled; it is consumed
 9   by  STORE
10
11
12
13
14
15
```

```
block 618
 0 ( 920910/vectors)  HEX
 1  0   =@R0 1 ~ :V V1.01   16 + 2 ;
 2  0   =@R1 1 ~ :V V1.02   17 + 2 ;
 3  0   =R0  1 ~ :V V1.03   18 + 2 ;
 4  0   =R1  1 ~ :V V1.04   19 + 2 ;
 5  0   =R2  1 ~ :V V1.05   1A + 2 ;
 6  0   =R3  1 ~ :V V1.06   1B + 2 ;
 7  0   =R4  1 ~ :V V1.07   1C + 2 ;
 8  0   =R5  1 ~ :V V1.08   1D + 2 ;
 9  0   =R6  1 ~ :V V1.09   1E + 2 ;
10  0   =R7  1 ~ :V V1.10   1F + 2 ;
11 =@R0  0   1 ~ :V V1.11   36 + 2 ;
12 =@R1  0   1 ~ :V V1.12   37 + 2 ;
13 =R0   0   1 ~ :V V1.13   38 + 2 ;
14 =R1   0   1 ~ :V V1.14   39 + 2 ;
15

block 619
 0 ( 920910/vectors)  HEX
 1 =R2  0   1 ~ :V V1.15   3A + 2 ;
 2 =R3  0   1 ~ :V V1.16   3B + 2 ;
 3 =R4  0   1 ~ :V V1.17   3C + 2 ;
 4 =R5  0   1 ~ :V V1.18   3D + 2 ;
 5 =R6  0   1 ~ :V V1.19   3E + 2 ;
 6 =R7  0   1 ~ :V V1.20   3F + 2 ;
 7 =A  =@R0 1 ~ :V V1.21   76 + 1 ;
 8 =A  =@R1 1 ~ :V V1.22   77 + 1 ;
 9 =A  =R0  1 ~ :V V1.23   78 + 1 ;
10 =A  =R1  1 ~ :V V1.24   79 + 1 ;
11 =A  =R2  1 ~ :V V1.25   7A + 1 ;
12 =A  =R3  1 ~ :V V1.26   7B + 1 ;
13 =A  =R4  1 ~ :V V1.27   7C + 1 ;
14 =A  =R5  1 ~ :V V1.28   7D + 1 ;
15

block 620
 0 ( 920910/vectors)  HEX
 1 =A   =R6 1 ~ :V V1.29   7E + 1 ;
 2 =A   =R7 1 ~ :V V1.30   7F + 1 ;
 3 =@R0 =A  1 ~ :V V1.31   86 + 1 ;
 4 =@R1 =A  1 ~ :V V1.32   87 + 1 ;
 5 =R0  =A  1 ~ :V V1.33   88 + 1 ;
 6 =R1  =A  1 ~ :V V1.34   89 + 1 ;
 7 =R2  =A  1 ~ :V V1.35   8A + 1 ;
 8 =R3  =A  1 ~ :V V1.36   8B + 1 ;
 9 =R4  =A  1 ~ :V V1.37   8C + 1 ;
10 =R5  =A  1 ~ :V V1.38   8D + 1 ;
11 =R6  =A  1 ~ :V V1.39   8E + 1 ;
12 =R7  =A  1 ~ :V V1.40   8F + 1 ;
13 =@R0 =#  1 ~ :V V1.41    6 + 2 ;
14 =@R1 =#  1 ~ :V V1.42    7 + 2 ;
15

block 621
 0 ( 920910/vectors)  HEX
 1 =R0  =# 1 ~ :V V1.43    8 + 2 ;
 2 =R1  =# 1 ~ :V V1.44    9 + 2 ;
 3 =R2  =# 1 ~ :V V1.45   0A + 2 ;
 4 =R3  =# 1 ~ :V V1.46   0B + 2 ;
 5 =R4  =# 1 ~ :V V1.47   0C + 2 ;
 6 =R5  =# 1 ~ :V V1.48   0D + 2 ;
 7 =R6  =# 1 ~ :V V1.49   0E + 2 ;
 8 =R7  =# 1 ~ :V V1.50   0F + 2 ;
 9  0    0 1 ~ :V V1.51   15 + 3 ;
10  0   =# 1 ~ :V V1.52    5 + 3 ;
11  0   =A 1 ~ :V V1.53   85 + 2 ;
12  0   =C 1 ~ :V V1.54   22 + 2 ;
13 =A    0 1 ~ :V V1.55   75 + 2 ;
14 =C    0 1 ~ :V V1.56   32 + 2 ;
15

block 622
 0 ( 920910/vectors)  HEX
 1 =A    =# 1 ~ :V V1.57    4 + 2 ;
 2 =DPTR =# 1 ~ :V V1.58   20 + 3 ;
 3
 4 =A =@R0 2 ~ :V V2.11    6 + 1 ;
 5 =A =@R1 2 ~ :V V2.12    7 + 1 ;
 6 =A =R0  2 ~ :V V2.13    8 + 1 ;
 7 =A =R1  2 ~ :V V2.14    9 + 1 ;
 8 =A =R2  2 ~ :V V2.15   0A + 1 ;
 9 =A =R3  2 ~ :V V2.16   0B + 1 ;
10 =A =R4  2 ~ :V V2.17   0C + 1 ;
11 =A =R5  2 ~ :V V2.18   0D + 1 ;
12 =A =R6  2 ~ :V V2.19   0E + 1 ;
13 =A =R7  2 ~ :V V2.20   0F + 1 ;
14  0 =#   2 ~ :V V2.21    3 + 3 ;
15

block 623
 0 ( 920910/vectors)  HEX
 1  0 =A 2 ~ :V V2.22    2 + 2 ;
 2 =A  0 2 ~ :V V2.23    5 + 2 ;
 3 =A =# 2 ~ :V V2.24    4 + 2 ;
 4 =C  0 2 ~ :V V2.25   32 + 2 ;
 5 ( =C =/ 2 ~ :V V2.26   60 + 2 ;)
 6
 7 =@R0 0 3 ~ :V V3.01    6 + 1 ;
 8 =@R1 0 3 ~ :V V3.02    7 + 1 ;
 9 =R0  0 3 ~ :V V3.03    8 + 1 ;
10 =R1  0 3 ~ :V V3.04    9 + 1 ;
11 =R2  0 3 ~ :V V3.05   0A + 1 ;
12 =R3  0 3 ~ :V V3.06   0B + 1 ;
13 =R4  0 3 ~ :V V3.07   0C + 1 ;
14 =R5  0 3 ~ :V V3.08   0D + 1 ;
15

block 624
 0 ( 920910/vectors)  HEX
 1 =R6   0 3 ~ :V V3.09    0E + 1 ;
 2 =R7   0 3 ~ :V V3.10    0F + 1 ;
 3  0    0 3 ~ :V V3.11     5 + 2 ;
 4 =A    0 3 ~ :V V3.12     4 + 1 ;
 5 =DPTR 0 3 ~ :V V3.13   0A3 + 1 ;
 6
 7 =R0 0 4 ~ :V V4.01    8 + 2 ;
 8 =R1 0 4 ~ :V V4.02    9 + 2 ;
 9 =R2 0 4 ~ :V V4.03   0A + 2 ;
10 =R3 0 4 ~ :V V4.04   0B + 2 ;
11 =R4 0 4 ~ :V V4.05   0C + 2 ;
12 =R5 0 4 ~ :V V4.06   0D + 2 ;
13 =R6 0 4 ~ :V V4.07   0E + 2 ;
14 =R7 0 4 ~ :V V4.08   0F + 2 ;
15  0  0 4 ~ :V V4.09    5 + 3 ;

block 625
 0 ( 920910/vectors)  HEX
 1 =@R0 =# 5 ~ :V V5.01    6 + 3 ;
 2 =@R1 =# 5 ~ :V V5.02    7 + 3 ;
 3 =R0  =# 5 ~ :V V5.03    8 + 3 ;
 4 =R1  =# 5 ~ :V V5.04    9 + 3 ;
 5 =R2  =# 5 ~ :V V5.05   0A + 3 ;
 6 =R3  =# 5 ~ :V V5.06   0B + 3 ;
 7 =R4  =# 5 ~ :V V5.07   0C + 3 ;
 8 =R5  =# 5 ~ :V V5.08   0D + 3 ;
 9 =R6  =# 5 ~ :V V5.09   0E + 3 ;
10 =R7  =# 5 ~ :V V5.10   0F + 3 ;
11 =A   =# 5 ~ :V V5.11    4 + 3 ;
12 =A    0 5 ~ :V V5.12    5 + 3 ;
13
14
15

block 626
 0 ( 920910/vectors)  HEX
 1  0 0 6 ~ :V V6.01    2 + 2 ;
 2 =A 0 6 ~ :V V6.02   44 + 1 ;  ( 44 for A CPL ; 24 for A CLR )
 3 =C 0 6 ~ :V V6.03    3 + 1 ;
 4
 5  0 0 7 ~ :V V7.01   3 ;
 6
 7  0 0 8 ~ :V V8.01   2 ;
 8
 9  0 0 9 ~ :V V9.01   3 ;
10
11 ( 0 0 0A ~ :V VA.01 ;)
12
13
14
15
```

On the Back Burner #4

Some Assembly Required...

Conducted by Russell L. Harris
Houston, Texas

As promised, with this column we begin an expedition into the realm of embedded systems. According to the ancient proverb, a journey of a thousand miles begins with a single step. Our first step, as you will shortly see, is directly onto a figurative "cow pie." (Those of you unfamiliar with the term *obviously* have never walked through a pasture in which cattle graze.)

A Rational Rationale

The nature of Forth, as well as the nature of embedded systems, necessitates the occasional use of assembly language. Although hand assembly is possible, it is tedious and prone to error. An assembler is almost always a worthwhile investment. Also, designing and coding an assembler is one of the better ways to gain familiarity with the instruction set of a processor.

While it is possible to utilize an assembler which is external to the Forth environment, the convenience of an assembler integrated with Forth and the ease (in general) with which such a tool may be created, combine to make the writing of assemblers a fairly common activity among Forth programmers. Forth programmers experienced in metacompilation typically will write an assembler upon first encountering a new processor. The assembler then can serve both as the means to port Forth to the new platform and as the resident assembler for the new Forth system.

There are processors for which this task can become an arduous and irksome chore.

Consistent Inconsistency

The art of assembler design admits of many interpretations. I find most appealing the approach of Forth, Inc., as illustrated by the 8080 assembler in *Starting Forth*. The source for polyForth assemblers I have seen typically occupies less than half a dozen screens. However, such compactness is possible only when the processor instruction set consistently follows patterns.

If a processor has a reasonably consistent instruction set, an assembler is neither a lengthy nor a difficult undertaking. However (and here is where the cow pies come in), there are processors for which the task can become an arduous and irksome chore, rather than a stimulating exercise. Such, unfortunately, is the case for the 8051 processor family, the family with which we shall deal. The 8051 instruction set is a hodgepodge, difficult to handle by any means.

Seeing an upcoming need (that of a potential client) for a Forth system for the 8051 family, I decided to assault two birds with one stone—hence, our project: an 8051 assembler. Were my client not already committed to the 8051 family, our present and future endeavours would be based on a Motorola processor, such as the 68HC11. However, I cannot at present manage a parallel effort with both platforms, so, unless some patron wishes to rescue us by engaging my services for programming in the Motorola environment, we are doomed to the wastelands of Intel. Circumstances such as this have left our civilization burdened with such ill-conceived contrivances as the segmented memory architecture of the 80x86, the QWERTY keyboard, and Word Perfect. But then, that's life. (Note: The author types on a Dvorak keyboard and does all his writing with Microsoft Word.)

The Nitty-Gritty

The accompanying screens contain the basis of an extensible 8051 assembler which, in its present state, compiles all 8051-family instructions, except for a couple of pathological cases. The assembler is written in polyForth ISD-4/MS-DOS for the 8086/8088. An entire instruction is built on the stack before being compiled. Included in the code is support for a virtual array on disk, into which the assembled code may be compiled. The assembler uses postfix notation, and operands must be separated by spaces, rather than by commas. Otherwise, the opcode mnemonics and operands are as specified in the appropriate Intel documentation. Some examples of valid syntax are the following instructions:

```
@A+DPTR JMP          A # 25 XRL
A 32 41 CJNE         @R0 # 57 2 CJNE
5 C MOV              C 6 MOV
```

The assembler is based on active operands, a table of execution vectors, and a mechanism (a toggle and a two-byte array) for tagging the first operand encountered. Named operands (#, A, @R0, R1, etc.) are active, in the sense of having a run-time behaviour other than that of CONSTANT. Execution of a named operand loads either the first or the second byte of the array OPERANDS with the value of the operand and sets the toggle SEQUENCE. The toggle initially is clear, and is cleared after assembly of an instruction. If the toggle is clear, the operand value is stored in the first byte of OPERANDS; if the toggle is set, the value is placed in the second byte.

A problem not initially envisioned was the need to discriminate between instructions of the form

(named operand)(numeric operand)(mnemonic)

and those of the form

(numeric operand)(named operand)(mnemonic)

without requiring non-standard syntax. When parsing the input stream, the text interpreter automatically converts numeric operands (i.e., data or address bytes) and pushes them onto the stack; thus, with no flag or mechanism to indicate that a numeric operand precedes it, the named operand always stores its value into the first byte of OPERANDS.

In an effort to avoid redesign of the entire assembler, I envisioned two approaches to the problem. The first was to parse the input stream under program control, then attempt to convert the resulting string, using CONVERT (because CONVERT returns an address which can be used to determine success of the conversion). Successful conversion would automatically push numeric operands onto the stack. A string which failed to convert would be either a named operand or a mnemonic. In such a case, juggling of the input pointer >IN could allow the string to again be parsed and then executed. I experimented for a while with this technique, but was unable to devise a suitable implementation, so I turned to the second approach, which was to redefine NUMBER.

Upon loading the assembler, I revector NUMBER to a version which, after performing a conversion, sets the toggle. Thus, encounter of a numeric operand causes the following named operand, if any, to place its value in the second byte of OPERANDS. This solution does not interfere with the ordinary function of NUMBER, but there is an associated hazard, as detailed in the shadow block documentation.

I have grouped the 8051 instructions into classes, in which all members of a class follow the same pattern with respect to operands. Instruction mnemonics (ADD, SUBB, XRL, etc.) are defined with :INSTRUCTION. When executed, a mnemonic pushes onto the stack the basis or base value for the opcode and calls ASSEMBLE. ASSEMBLE uses the instruction class and operand numbers to index into the three-dimensional array VECTORS in order to obtain an execution vector. The typical run-time behaviour of a vector is to add to the base value an offset corresponding to the operand(s), then push onto the stack the number of bytes to be compiled.

After ASSEMBLE executes the vector associated with a particular combination of class and operands, control passes to PREPARE, which clears both the toggle and the array OPERANDS. Control passes thence to the vectored routine STORE, which disposes of the assembled code, now resident on the stack. By default, STORE is vectored to DISPLAY, which simply displays and then clears the stack. STORE may be redirected to VSTORE in order to compile the 8051 code into a virtual array on disk. It is a simple matter to redirect STORE to other destinations, e.g., a serial port.

Note the ease with which the virtual array is implemented: a single source block does it all! The same approach may be used for a virtual array in extended memory. Virtual memory techniques are invaluable for data logging applications, and they form the basis of metacompilation.

An understanding of defining words is essential to the mastery of Forth. Note the nesting of the defining words :CLASS and :INSTRUCTION. Also note the manner in which the defining words :O and :V are used to define operands and vectors, respectively. In spite of its unusual appearance, operation of the defining word :V is really quite simple. :V is nothing more than a : which calculates the PFA of the word being defined and stores the PFA into the array VECTORS. Otherwise, :V is used as one would use :.

Although the assembler is usable in its present state, several amenities remain to be added, among them, labels and high-level Forth control of loops and branching. Also, at the cost of creating a separate class for each instruction (thus expanding the array VECTORS), it should be possible to trap all invalid combinations of operand and mnemonic.

This code will be posted on GEnie. If there is sufficient interest, I will post an updated listing once my implementation is complete. Conversely, I am interested to see what my readers do, given this code as a basis or for inspiration.

Preview of Coming Attractions

For the next leg of our journey, you may want to pull out your soldering iron and wire-wrap tool. Metacompilation and related subjects are easier to discuss and understand when specific instances are in view. Accordingly, column No. 5 will complete the preliminaries by documenting a reproducible, minimal-cost, 8032-based single-board computer (SBC). Boasting little more than a serial and a parallel port, a reset button, and a full complement of RAM, the device is an easy weekend project in the $50 range. It has been designed for software development in RAM, and requires neither EPROM programmer nor ROM emulator. For those with an aversion to hardware projects, I will attempt to find a source of a suitable commercial SBC.

R.S.V.P.

Russell Harris is an independent consultant providing engineering, programming, and technical documentation services to a variety of industrial clients. His main interests lie in writing and teaching, and in working with embedded systems in the fields of instrumentation and machine control. He can be reached by phone at 713-461-1618 or by mail at 8609 Cedardale Drive, Houston, Texas 77055. Caveat: His GEnie address is RUSSELL.H).

"A rose by any other name would still have thorns."

Code begins on page 33, and can also be downloaded from the Forth RoundTable on GEnie.

Fourteenth Annual

FORML CONFERENCE

The original technical conference
for professional Forth programmers, managers, vendors, and users.

Following Thanksgiving, November 27 – November 29, 1992

Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California U.S.A.

Theme: Image display, capture, processing, and analysis

Papers are invited that address relevant issues in the development and use of Forth in image display, capture, processing, and analysis. Additionally, papers describing successful Forth project case histories are of particular interest. Papers about other Forth topics are also welcome.

Conference Registration

Registration fee for conference attendees includes conference registration, coffee breaks, notebook of papers submitted, and for everyone rooms Friday and Saturday, all meals including lunch Friday through lunch Sunday, wine and cheese parties Friday and Saturday nights, and use of Asilomar facilities.

Conference attendee in double room—$365 • Non-conference guest in same room—$225 • Children under 18 years old in same room—$155 • Infants under 2 years old in same room—free • Conference attendee in single room—$465

Forth Interest Group members and their guests are eligible for a ten percent discount on registration fees.

Register by calling the Forth Interest Group business office at 510-893-6784 or by writing to:

FORML Conference, Forth Interest Group, P.O. Box 2154, Oakland, CA 94621

Forth Interest Group
P.O. Box 2154
Oakland, CA 94621

Second Class
Postage Paid at
San Jose, CA