$10 Volume XV, Number 3 September 1993 October

# FORTH DIMENSIONS

# Contents

## Features



## Departments

# Editorial

## Why FIG Chapters?

The newly formed Southern Wisconsin chapter of the Forth Interest Group describes what happened at one of their first meetings:

"After about half an hour of conversation, Dave Ruske spoke about the use of Forth in two embedded applications for ICOM, Inc. The first was an 8031-based operator access panel for an Allen-Bradley PLC-2; the second was a driver that loads onto the Allen-Bradley KT card (a Z80-based network card for PLC communications). LMI metacompilers were used on both projects.

"Dr. Bob Lowenstein of Yerkes Observatory told the group about Yerk and how it is applied to telescope control and data acquisition. Yerk is an object-oriented Forth variant for the Mac, a public-domain product derived from Neon. One of the things Dr. Lowenstein uses this for is remotely controlling a telescope in New Mexico via the Internet. Yerk software will also be used to control two new telescopes being built at the South Pole.

"Matt Mercaldo showed and talked about the Modular Microprocessor Trainer being developed for use at Johns Hopkins University. The unit uses two Motorola MC68HC11A8s with New Micros' Max-Forth on chip. The student can prototype projects on the unit and communicate with it using a terminal, or with a keyboard and built-in LCD. Matt provided the group with copies of a paper describing his work.

"Glenn Szejna described his use of Forth at Nicolet Instruments. Several people in the group had used Nicolet oscilloscopes, unaware that Glenn's Forth code was running under the hood.

"Scott Woods discussed his use of Forth, including his current project, firmware for an industrial metal detector. This device will be used for such things as preventing machine screws from showing up in your breakfast cereal.

"Olaf Meding described Amtelco's use of polyForth in programming their systems for telephone answering services [see "Forth-Based Message Service," *Forth Dimensions* XIV/5]. The Amtelco EVE (Electronic Video Exchange) is the largest and most sophisticated system of its kind, and has gained 70% of the answering service market.

"James Heichek demonstrated VORCOMP, a public-domain directory and file-compare utility written in his own version of Forth. James talked about why he believed the stack manipulation words in Forth became a hindrance to his work, and about how he added local variables to clean up his code. He is currently developing educational software.

"Olaf Meding gave a brief tour of the 'Introduction to Forth' disk and demonstrated loading C.H. Ting's tutorial in F-PC. Along the way, the F-PC single-step debugger was also demonstrated.

"[As] the meeting began to break up, Dr. Lowenstein took some time to demonstrate Yerk and to impart some insider information to Paul Anderson. Paul is new to Forth and plans to build a system to monitor and control model railroads via a Mac."

Thanks to Dave Ruske for providing these notes, and congratulations to the Southern Wisconsin FIG Chapter for forming what is obviously a dynamic and resource-rich group. It was only a few months earlier that these folks contacted the FIG office for a Chapter kit. If there is no FIG Chapter in your area, perhaps you should think of doing the same...

Regularly scheduled meetings; guest lecturers; planned presentations and demonstrations; advance mailing of meeting details; and plenty of interaction, flexibility, and attention to the particular needs and interests of your group—perhaps including occasional tutorials or ongoing classes for Forth neophytes—are elements that keep chapter meetings lively and useful. If there is already a chapter near you, think about how your added participation might help that group, then talk to the chapter's officers and organizers about volunteering—whether as stamp-licker, secretary, emcee, program chairman, snack officer, or next chapter president.

Who knows, maybe you'll even discover—like at least one Forth author—that steady Forth-related employment is a fringe benefit of the contacts made at your local FIG Chapter.

*—Marlin Ouverson*

**Forth Dimensions**
Volume XV, Number 3
September 1993 October

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, c/o TPI, 1293 Old Mt. View-Alviso Rd., Sunnyvale, CA 94089. 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!*

### Ready to Pay

Dear Marlin:

Thanks to Markus Dahm for contributing the article about natOOF in your last issue (*FD* XV/2). I've dabbled in Forth since I was a hardware engineer, and this development is exciting for a lot of reasons. My current work involves developing presentations and visual interfaces. I use SuperTalk, HyperTalk, OpenScript, and, if I have to, C++. The first two run on the Mac, the third on the PC. All three are very similar to each other and, it turns out, to natOOF.

The concept of natOOF is exciting because it means that a scripting language based on Forth is being created that has a good chance of wide and profitable acceptance by the public and by the software development community. It is the ideal way to bypass the engineers and managers who don't understand Forth, and to take it directly to the public. It is also a good platform for a visual programming language and for visual interfaces in general.

It is particularly exciting to me because I can have a simple scripting language, natOOF, based on a simple machine model, Forth. This means that, unlike the other scripting languages, I will be able to easily create new methods and whatever classes of objects I need. I will also have access to the underlying architecture of the machine if I need it. This is particularly helpful when tracking down the memory problems that come up (scripting languages still seem to mess up memory handling).

***I use Forth quite frequently for manufacturing automation, with good results. The language is very practical for our application...***

Those other scripting languages provide the user with a limited, albeit flexible, set of classes. Until recently, any compilation had to be done by creating XCMDs or DLLs in another language, usually C. Heizer Software now provides a product that allows you to create XCMDs using HyperTalk itself; however, one cannot create new classes or modify old classes.

I find that I can do projects in days and weeks, instead of months, by using scripting languages. With Forth as a platform, this will be even better. I am ready to pay money for natOOF now. When can I get it?

Sincerely,
Mark Martino
14115 NE 78th Court
Redmond, WA 98052

### More Good Results with Forth

Dear Mr. Ouverson:

I am not in the habit of writing letters to the editor, but the excellent article by Donald Kenney ("Forth in Search of a Job," *FD* XV/1) has prompted me to do it.

The attached memo *[not printed here—Ed.]* resulted from the fact that the customer, another division of the same corporation, wanted to use Forth in an automotive controller we were designing. The project ended up using a dialect of Modula-2.

Now working for a different corporation, I use Forth quite frequently for manufacturing automation, with good results. The language is very practical for our application (STD-based controllers for various machines), thanks to its ease and speed of coding, speed of execution, and elimination of the need for logic state analyzers, in-circuit emulators, and all the other rather expensive peripherals required with most high-level languages. Very few programmers are involved (a must with Forth, in my view), and the typical application requires only about 500 lines of code.

We use a PC for development, and transfer the code over an RS-232 port to the PROM-based controller, which compiles it. The same port is used to communicate with the running application during debugging. After release, the controllers work in standalone mode (no PC); if anything needs to be changed, or if we need to intervene in order to facilitate mechanical adjustments, we take a laptop up to the controller and connect it to the port, making it interactive again.

A battery-backed RAM (Dallas Semiconductor) in the program memory chip socket during code development eliminates having to wait for the program to reload every time the power is turned off. After release, a PROM can replace that RAM if frequent changes are not anticipated. In one controller, where some variables may need to be preserved after a reset, the program is allowed to allocate battery-backed program memory for those variables, thanks to the versatility of the system.

Sincerely yours,
Ivan C. Coelho
Manufacturing Engineering Manager
Joslyn Electronic Systems
6868 Cortona Drive
Goleta, California 93116-0817

# Character Classification

## Reverse Engineering a C Utility

*Charles Curley*
*Gillette, Wyoming*

A simple means of allowing applications programmers to classify characters (printable, control, upper case, etc.) is shown.

### Historical Note

The version of Forth used here is fastForth, a 68000 JSR/BSR-threaded Forth described in "Optimization Considerations," *Forth Dimensions* XIV/5. There is minimal code specific to fastForth, and implementation on other Forths should be fairly easy. `F@` is a fast, word-boundary-only version of `@`. Users of other Forths will have to adjust the code.

### Background

While implementing Martin Schaaf's "Formatted Input Fields" (*Forth Dimensions* XV/1), I examined the code to see how it was done. The code depended on three words for detecting the classification of a character. These are: `ALPHA`, `ALPHA-NUMERIC`, and `NUMERIC`. I thought the code for these words was a bit crude and could be better done. What was needed, I thought, was a much more flexible system which could also be much faster. A look-up table would be just the thing.

***When I learned C, the first thing I did was cure one of its lacks.***

Having worked in C, I knew exactly where to find such a lookup table: C's library routines and macros as described in the header file ctype.h. I decided to implement the C routines in Forth. In the words of Arlo Guthrie, I'm not proud. Or tired. Converting the routines and macros to Forth was easy. Duplicating the table was also easy, but only because I have experience with Forth and insisted years ago on curing at least one of the many problems with C.

Yes, folks, I admit it: I actually found something done better in C than in a Forth program. So one of the questions I had was whether I could do a better job in Forth than the implementors of my C compiler had done in C.

### Reverse Software Engineering

The first thing I did was examine the header file ctype.h. This consists almost entirely of a series of macro definitions (#defines, in C-ese) for use with a lookup table called _ctype[]. (For those of you who don't speak C, that describes a character array. On the 68000, that is a byte array as well. The reason for the _ (underscore) in the name has to do with the kinky requirements of C linkers.)

The first group of macros defines a series of bit masks for use with the lookup table at _ctype. For example, a bit mask to identify upper-case characters might be defined as follows:

```
#define      _U     01
/* Upper-case alphabetic */
```

The second group defines a series of macros to classify characters, using _ctype and the bit masks in the first group. For example, to determine if a character is upper case, use isupper(), which might be defined as follows:

```
#define      isupper(c)
(_ctype[(c)+1]&_U)
```

isupper() uses the character to be classified as an index into the table at _ctype to fetch a bit mask. This is anded bitwise (the ampersand operator) with the mask defined above, _U, to produce a flag. If bit 0 is set in that entry in the table, a result of 1 is returned by the macro, indicating that the test character is upper case. If bit 0 is reset, the character under test is not upper case.

Examination of the macros indicated one place where an improvement could be made: the index into the array is defined as (c)+1, which would involve at least one instruction to increment c. Why the table begins at _ctype+1 instead of at _ctype itself is unknown. The reason is most likely lost among the antiquities of Bell Labs, along with the programming languages A and B, and the Bell Labs version of the Ark of the Covenant. This instruction can be eliminated by starting the table at _ctype, so that is how the table will be implemented in Forth.

It would be possible to build the table at _ctype by determining each character's mask, and building it by hand. Instead, I chose to retain compatibility with C by carrying over the table in its entirety. Not having source for the table

```
                                                        Scr #  1920
 0 \ formatted input fields                    ( 21  5 93 CRC 18:07 )
 1 \ from article by Martin Schaff, 5,6 93 Forth Dimensions
 2 forget task forth definitions   : task ;
 3
 4
 5 base f@ >r
 6
 7
 8 1 12 +thru
 9
10
11
12
13 r> base f!      editor flush
14
15


                                                        Scr #  1921
 0 \ ctype character classifier                ( 21  5 93 CRC 18:07 )
 1 create _ctype                    hex
 2 \  0    1       2       3       4       5       6       7
 3 40 c,   40 c,   40 c,   40 c,   40 c,   40 c,   40 c,   40 c,
 4
 5 \  8    9       0a      0b      0c      0d      0e      0f
 6 40 c,   50 c,   50 c,   50 c,   50 c,   50 c,   40 c,   40 c,
 7
 8 \ 10    11      12      13      14      15      16      17
 9 40 c,   40 c,   40 c,   40 c,   40 c,   40 c,   40 c,   40 c,
10
11 \ 18    19      1a      1b      1c      1d      1e      1f
12 40 c,   40 c,   40 c,   40 c,   40 c,   40 c,   40 c,   40 c,
13 ;s      This is a lookup table for examining characteristics
14 of characters: alpha, numeric, printable, etc.
15


                                                        Scr #  1922
 0 \ ctype character classifier                  ( 21  5 93 CRC 18:07 )
 1 \ bl 20 !       "       #       $       %       &       '
 2 90 c,   20 c,   20 c,   20 c,   20 c,   20 c,   20 c,   20 c,
 3
 4 \ (  28 )       *       +       ,       -       .       /
 5 20 c,   20 c,   20 c,   20 c,   20 c,   20 c,   20 c,   20 c,
 6
 7 \ 0  30 1       2       3       4       5       6       7
 8  8 c,    8 c,    8 c,    8 c,    8 c,    8 c,    8 c,    8 c,
 9
10 \ 8  38 9       :       ;       <       =       >       ?
11  8 c,    8 c,   20 c,   20 c,   20 c,   20 c,   20 c,   20 c,
12
13 ;s      The table and its usage are patterned after the C
14 library routines for the same purpose. See ctype.h for more
15 details.


                                                        Scr #  1923
 0 \ ctype character classifier                  ( 21  5 93 CRC 18:07 )
 1 \ @  40 A       B       C       D       E       F       G
 2 20 c,    1 c,    1 c,    1 c,    1 c,    1 c,    1 c,    1 c,
 3
 4 \ H  48 I       J       K       L       M       N       O
 5  1 c,    1 c,    1 c,    1 c,    1 c,    1 c,    1 c,    1 c,
 6
 7 \ P  50 Q       R       S       T       U       V       W
 8  1 c,    1 c,    1 c,    1 c,    1 c,    1 c,    1 c,    1 c,
```

(one of the disadvantages of C libraries), I had to reverse engineer it.[1]

Fortunately, I had a ready solution to that problem. One of the first things I did when I determined to learn C was to cure one of its lacks. I wrote a memory dump routine, with the idea that I could build it into programs and get at least some ability to see memory modified as the program progressed. The syntax is: dump(addr*,len). Forthlike, isn't it?

An incredibly simple C program printed out a hex dump of the array at _ctype. Using operating system indirection, I wrote the output to a file. I then referred to the output file from time to time as I edited Forth screens to implement the table in Forth.

### The Forth Code

The Forth code is included in the fastForth implementation of Mr. Schaaf's code, starting on screen 1920. This screen is a loader screen, and could contain a vocabulary declaration, if one wanted to hide the code from the user.

Screen 1921 begins the lookup table with the phrase `create _ctype` on line one. Each character position has one hex value installed in it with the word `c,`. Above it is a comment indicating the character that the entry represents. For example, position zero, indicating ASCII value 0, has a bit mask of hex 40, indicating a control character. The tab character, ASCII 9, is both a control character and a white-space character, so its bit mask is hex 50. The table continues through screen 1924.

The code that uses the table is on the next screen, in the word `CLASSIFIER`. This defining word builds a classifier

1. The names have been changed to protect the publishers.

word, analogous to the second group of macros in ctype.h. On line five is code to build a header and comma a bit mask into the dictionary (see the stack diagram on line 13).

Also on that line, and commented out, is a high-level DOES> portion of the code. The code takes a value as input, leaving the input value and output flag on the stack (see the stack diagram on line 14).

The assembly language version is specific to the 68000 and to fastForth. On line six, the address of _ctype is loaded into address register zero. Line seven sees the return stack popped into address register one. This leaves the return stack containing the address of the word which called the classifier daughter word. It also leaves the address of that daughter word's argument in ar1. The next instruction reads the mask into data register zero.

On line eight, the code copies the character to be tested into data register one. A commented-out instruction would provide some range checking, emulating the C macro toascii(). The mask for the and is a 32-bit word value because only that portion of the register is used as an index.

Line nine provides one instruction, anding the bit mask in dr0 with the contents of the lookup table. The source-addressing mode is called "address register indirect with index." It adds the contents of address register zero to the word contents of data register one, and a sign-extended byte displacement (here, zero) to produce the effective address of the source. Assuming that data register one contains a value between zero and 7f,

```
 9
10 \ X  58 Y       Z       [       \       ]       ^       _
11  1 c,    1 c,    1 c,   20 c,   20 c,   20 c,   20 c,   20 c,
12
13
14
15
```

```
                                                      Scr #  1924
 0 \ ctype character classifier                ( 21  5 93 CRC 18:07 )
 1 \ @  60 a       b       c       d       e       f       g
 2 20 c,    2 c,    2 c,    2 c,    2 c,    2 c,    2 c,    2 c,
 3
 4 \ h  68 i       j       k       l       m       n       o
 5  2 c,    2 c,    2 c,    2 c,    2 c,    2 c,    2 c,    2 c,
 6
 7 \ p  70 q       r       s       t       u       v       w
 8  2 c,    2 c,    2 c,    2 c,    2 c,    2 c,    2 c,    2 c,
 9
10 \ x  78 y       z       {       |       }       ~       del
11  2 c,    2 c,    2 c,   20 c,   20 c,   20 c,   20 c,   40 c,
12
13
14
15
```

```
                                                      Scr #  1925
 0 \ ctype character classifier                ( 21  5 93 CRC 18:07 )
 1 \       Note the lack of range checking!
 2 \       Over 7f, you're on your own, tovarish!
 3
 4 : classifier
 5   create c, \   does> >r  dup  _ctype + c@ r> c@ and ; ;s
 6   ;code  _ctype *+ ar0 lea,                \ addr of table
 7   rp [+ ar1 mov,  ar1 [ dr0 .b mov,        \ mask to and
 8   s [ dr1 mov, ( 7f # dr1 .w and, )        \ the character
 9   0 ar0 1 &D[ dr0 .b and,                  \ and the mask
10   dr0 s -[ mov,                            \ results on the stack.
11   next  ;c              ;s
12
13 compile:  mask ---   \ compile a mask after the daughter word.
14 run:      c ---  c fl  \ return flag as to class of char.
15
```

```
                                                      Scr #  1926
 0 \ ctype character classifier                ( 21  5 93 CRC 18:07 )
 1    1 classifier isupper
 2    2 classifier islower
 3 \  4 isn't used.
 4    8 classifier isdigit
 5   10 classifier isspace
 6   20 classifier ispunct
 7   40 classifier iscntrl
 8 \ 80 means printable but no other class
 9
10  1 2 or                 classifier isalpha
11 20 80  3  8 or or or    classifier isprint
12  1 2 10 or or           classifier isalnum
13 \ or roll your own...
14
15
```

*(Continues on page 38.)*

# UN*X Tools Used on the FSAT Project

## (or, "And Now For Something Completely Different") Part I

*Jim Schneider*
*San Jose, California*

At some point in the writing of a large project in Forth, the typical Forth programmer (if I am typical, and if there is such a thing as a typical Forth programmer) fires up a meta- or cross-compiler to translate the work into a standalone application. This works fine as long as the programmer hasn't done anything that is radically different from what the metacompiler writer had in mind. If the new project depends on new defining words, a different dictionary linkage structure, or a different way of handling code and data space, the metacompiler (at least those I've worked with) probably won't be able to handle it without modifications. Additionally, most metacompilers use the host dictionary to save quite a bit of state information, so a metacompiler can't be used for a huge project if the host dictionary is very limited. These are my two primary reasons for using the UN*X toolset to do much of the development for the project.

To ensure that everything I write about in this series of articles is understood, I'm going to describe several UN*X tools. My primary emphasis is going to be on the programming commands *m4(1)*, *yacc(1)*, and *lex(1)*, but I'm also going to touch briefly upon *sh(1)* and *make(1)*.

**The features discussed form a large enough subset to perform some useful work. I'm going to use these features (and only these) in later articles...**

*sh(1)*, the default command interpreter (or shell) on most UN*X systems, allows you to write programs that take advantage of the abilities of other programs. For example, this shell script (a script is a program to be interpreted by the shell, like an MS-DOS batch file) will write the contents of the files specified on its command line to the standard output, and provide a header and footer for each:

```
#!/bin/sh
while [ -n "$1" ]
do
echo " /********** start of $1 **********/"
cat $1
echo " /********** end of $1 **********/"
shift
done
```

This is a trivial example, but I used this script several times to download a small program from a remote UN*X site to my home box. More complicated scripts can automatically configure and install software, perform periodic backups, process huge text files and databases, or just about any other task. In fact, it is so useful that about the only reasons UN*X utilities are written in other languages is that the shell is expensive in terms of resources used versus results produced, and it doesn't manipulate binary files all that well.

Because the shell is good for general purpose odd jobs, but not for jobs that need to be done efficiently and repeatedly, UN*X has a large assortment of utility programs. These can be broken down into categories in a variety of ways, but probably the most useful categories are (traditional) programming tools, text manipulation utilities, and miscellaneous. The traditional programming tools include things like compilers, linkers, library creation and maintenance utilities, source code control packages, and a peculiar little utility to ensure programs are up to date. This "peculiar little utility" is called *make*.

*make* functions by reading a file called (what else?) a make file. The make file contains lines that describe which files depend on other files (dependency lines), and lines to tell *make* what to do if a file is out of date (i.e., a dependent is newer than something that depends on it, called a target). For example, the following fragment of a make file:

```
foo.o: foo.c foo.h Makefile
       cc -c $(CFLAGS) $*.c
```

tells *make* that if either `foo.c`, `foo.h`, or `Makefile` (usually the default name of the make file) were modified more recently than `foo.o`, it is to perform a macro substitution on the second line and then execute it. *make* distinguishes dependency lines from command lines by the fact that command lines start with one or more tab characters. One point to remember about dependency lines: they may contain any number of dependents, but only one target. Thus, this would be invalid:

```
foo.o bar.o: foo.c bar.c foo.h Makefile
```

An important feature of command lines is the fact that they can contain *any* valid UN*X commands.

Macro substitution is a form of text processing that substitutes any macro name references in the text by a previously defined substitution. There are two kinds of macros in *make*: built-in (the `$*` in the example) and explicit (the `$(CFLAGS)`). The built-in macros are set by *make* to stand for either targets or dependents of the last dependency line it read. Without going into too much detail, the `$*` in the example stands for `foo`, or the target without its extension. Explicit macros are set in the make file, in the environment, or are built into *make*. If *make* processed a line like this:

```
CFLAGS=-O -I./include
```

before it encountered the two lines in the example, the string `$(CFLAGS)` would be transformed by macro substitution into `-O -I./include`.

*make* executes a line by passing it to *sh* (hey, *sh* was designed to execute command lines). Thus, the sequence of events that *make* would follow when it encounters the example above would be:

1. Check to see if `foo.o` is out of date with respect to `foo.c`, `foo.h`, or `Makefile`
2. If `foo.o` is out of date, transform the line:
   ```
   cc -c $(CFLAGS) $*.c
   ```
   into:
   ```
   cc -c -O -I./include foo.c
   ```
   and pass the transformed line to *sh*. *cc(1)*, the UN*X C compiler, uses the arguments `-O` and `-I./include` as options. They mean: perform optimization and search the path `./include` for included files, respectively.

Another feature of *make* is its ability to use built-in dependency rules. These rules tell *make* how to transform one file type to another without explicit rules. For example, the following two lines:

```
.c.o:
        cc -c $(CFLAGS) $<
```

tell *make* that whenever it finds a dependency line with a target with an extension of `.o` and a dependent file with an extension of `.c` that has a dependent file newer than the corresponding target, it would have the second line of the rule executed by *sh*, after macro substitution, in the absence of explicit commands. The macro `$<` stands for the dependents that are newer than the target in the dependency line. This means that if *make* were to process this line:

```
foo.o: foo.c foo.h Makefile
```

without any line following it, and any of the dependents were newer than the target, *make* would pass this line to *sh*:

```
cc -c -O -I./include foo.c
```

The `.c.o` in the dependency line is called a suffix rule. The reason I used `$*.c` instead of `$<` in the first example was to handle the case where (for example) `Makefile` was the newer file. In such a case, *make* would pass the incorrect line:

```
cc -c -O -I./include Makefile
```

to *sh* to execute. Since `Makefile` is not a C language file, *cc* will issue unfriendly diagnostics, and return an error code to *make*. The reason this works in the built-in dependency rule is that the `.c` in the rule tells *make* to substitute only the names of dependents that have an extension of `.c`. It will substitute these names (and only these names) into the line, regardless of which file is newer than the target.

If there is only one suffix on the target line of the built-in dependency line, it is taken to be the extension of the dependent file, and the target is the dependent filename without the suffix. For example, this:

```
.c:
        cc $(CFLAGS) -o $@ $<
foo: foo.c
```

tells *make* that in order to build `foo` from `foo.c`, it is to perform macro substitution on the command line, and pass it to *sh*. The built-in macro `$@` stands for the current target file. Assuming that the macro `CFLAGS` is defined as above, the line will be transformed into:

```
cc -O -I./include -o foo foo.c
```

(*cc* uses the option `-o` to override the default name of its output file.)

At this point, I should mention a caveat. The suffix rule in the built-in dependency must be "registered" by *make*. To register a suffix rule, it has to be added to the built-in macro `.SUFFIX`. For example, if you want to build `.s` files from `.m` files, before you could put this:

```
.m.s:
        m4 $< > $@
```

into a make file, you would probably need to put this:

```
.SUFFIX=$(.SUFFIX) .m.s
```

into the make file first. Usually, these suffix rules are built into the macro `.SUFFIX`:

```
.c     .sh    .c.o   .c.a   .s.o
.y.o   .l.o   .y.c   .l.c
```

The UN*X operating system uses the suffixes `.c`, `.sh`, `.o`, `.a`, `.s`, `.y`, and `.l` for C language source files, shell scripts, object files, libraries, assembly langauge source files, *yacc* language source files, and *lex* language source files, respectively. Additionally, the suffix `.h` is used for C language header files. I, myself, use the suffixes `.m` and `.i` for *m4* language source and include files, respectively.

An additional feature of *make* is the possibility of having more than one command line for a given dependency line. For example, the pseudotarget `clean` is often used to "clean up" after a project is built. `clean` is called a pseudotarget because its commands don't actually create or update a file with the name `clean`. `clean` may have a dependency line and several command lines, like:

```
clean:
        rm -f *.o core a.out
```

```
	cd target1 ; rm -f *.o core a.out
	cd target2 ; rm -f *.o core a.out
```

In this case, clean has no dependents, so it is always considered to be "out of date." The three command lines in this example would be passed to *sh*, which would execute the commands. The reason that the second and third command lines contain two actual commands—*cd(1)* and *rm(1)*—has to do with a peculiarity of *make*. As each line is generated, *make* passes it to a fresh invocation of *sh*. Since each command line in a make file corresponds to a different shell, all commands that are built into *sh* (like *cd* in the example) have no effect after the end of the line.

Finally, a line starting with a # character is a comment.

To get *make* to update a specific target, invoke *make* with that target as an argument. If *make* is invoked without arguments, it checks the first target in the make file. For example, if this was the make file:

```
.c.o:
	cc -c $<
CFLAGS=-O -I./include
OBJS=foo.o bar.o baz.o
foo: $(OBJS)
	cc $(CFLAGS) -o $@ $(OBJS)
foo.o: foo.c foo.h Makefile
bar.o: bar.c foo.h Makefile
baz.o: baz.c Makefile
clean:
	rm -f *.o core foo
```

the command:

```
make baz.o
```

would tell *make* to check `baz.o`, while this:

```
make
```

would tell *make* to check `foo`.

You now know more about *make* than I did when I started using it. In any make file associated with this project, I will only use the features described above, and I will explicitly define all explicit macros. I will also explicitly define any built-in dependencies.

There are several programming tools that also fall into the text manipulation category. Indeed, many of the tasks we give to computers are in the realm of text manipulation; if we broaden our definition of text, they all are. UN*X-like operating systems have many utilities to make the job of custom text manipulation easier. The two main divisions of the text processing category relate to how they do the job. The first provides the ability to manipulate streams of text. This is typified by this fragment:

```
sed '/BEGIN/,/END/d' filex
```

which tells *sed* to read the file `filex`, and print all the lines in the file, but delete (from the output only, not the original file!) all the lines after the first occurrence of the string `BEGIN` (including the line it occurs on) up to the first following line that contains the string `END`. So, if this was the contents of filex:

```
This is the documentation for filex
(UN*X-type man page here)
BEGIN uuencoded image
(miscellaneous stuff here)
END uuencoded image
(more miscellaneous verbiage)
```

## UN*X Trivia

*For those of you who are curious, these are explanations of the names of all the UN*X utilities mentioned in this article (with the exception of m4).*

*yacc* — stands for "Yet Another Compiler Compiler." When the Unix system was young, almost everyone involved in the project was fascinated with advances in the field of compiler theory. When this utility was installed, it was the fifth or sixth in the series of programs that turn grammar descriptions into parsers.

*lex* — stands for "LEXical analyzer"

*sh* — stands for "SHell"

*make* — "makes" an update of a program

*cat* — "conCATenates" the contents of its arguments to the standard output

*cc* — "C Compiler"

*sed* — "Stream EDitor"

*grep* — from the *ed* vernacular "g/RE/p" which stands for "global scan for regular expression and print result"

*ed* — "line EDitor"

*awk* — stands for its authors' initials: Aho, Wienberger, and Kernighan. *awk* is a powerful, stream-oriented, regular-expression-based programming language

By the way, I refer to Unix-like operating systems by "UN*X" because:

1. There are several Unix and Unix-like operating systems available, and only two are actually named Unix. Since my primary development platform is a 386 PC clone with both XENIX and Linux (if I ever figure out how to make it stop fighting XENIX for the disk drives...), which are Unix-like operating systems, I use the construct UN*X to refer to both of them. I may aquire a Sun SPARCstation in the near future, which has yet another flavor of Unix (SunOS, a very good port of 4.?BSD Unix)

2. Unix was until recently a trademark owned by AT&T. It is now owned by Novell. Since I harbor no love for either organization, I'm going to use a name that neither of them owns.

the output would be:

```
This is the documentation for filex
(UN*X-type man page here)
(more miscellaneous verbiage)
```

This is another trivial example, but it is useful for processing, say, a file containing text, an encoded binary image, and then more text.

The second division of text processing is macro processing. *make* does this to generate lines to execute. The UN*X operating system also provides a macro processor called *m4*. (Don't ask me why—I could tell you why "*sed*" is named *sed*, or even why "*awk*" is named *awk*, but I have no idea why "*m4*" is named *m4*!) The syntax of *m4* is very basic. Each separate word in the input stream is matched against the symbol table. If a match is found, the symbol is replaced by the string it stands for, and the process starts again at the beginning of the substitution. If the word doesn't match a defined symbol, the word is copied to the output stream. (Note: because *m4* operates literally, and the placement of new lines is important, I will put a # character in the examples wherever a new line is supposed to be. So, even though a line may end at the end of a column in this article, if you are typing the examples into a computer, don't hit the enter key until you see the # character.) Thus, this fragment:

```
define(`jim',`is God')#
jim#
```

would print out

```
#
is God#
```

Notice the extra line in the output. This is because the new line at the end of the first line wasn't matched by a macro name, so it was printed on the standard output. Indeed, the new-line character can't match a macro name because *m4* macro names consist of an upper- or lower-case letter or underscore followed by any number of letters, digits, or underscores. Anything else is either copied verbatim to the standard output, or used as punctuation. Macros are defined as shown in the example. The macro `define` (yes, it's a macro, too) is called with two arguments. The first is the name of the new macro, and the second is what the new macro stands for. Also, notice the quotes around the name and the substitution of the macro. I used them because *m4* will do macro substitution of arguments to macros before the arguments are passed to the macro. This means that if *m4* were to encounter:

```
define(jim,isn't God)#
```

as the next line it were to process, it would perform macro substitution on `jim`, and pass the arguments `is God` and `isn't God` to `define`. `define` would see that its first argument doesn't look like a name, and either do nothing or cause *m4* to bail out. The net result would be printing another new line (or exiting *m4* with an error message). *m4* lets you get around this by quoting. When *m4* is in macro substitution phase and sees a string with quotes around it, it strips off the outermost set of quotes instead. Because quotes delay macro substitution, it's a good idea to quote the arguments to a macro. *m4* will consider a string of text that starts with a backquote (`) and ends with a single quote (') to be a single word.

*m4* provides other macros besides `define`. One of the most useful is `ifelse`. This macro takes any number of arguments, and if the first argument is equal to its second, the result is its third. If its first argument doesn't match its second, and it has enough arguments, it tries to match its fourth and fifth, seventh and eighth, etc. If none of the pairs it checks match, and the last argument is its fourth, seventh, etc., that is its result. If none of these cases work, the result is the null string.

All of this is much easier to do than to explain. For example, this:

```
ifelse(cat,dog,`cat equals dog',
`cat does not equal dog')#
```

would either print out `cat equals dog`, or `cat does not equal dog`, depending on whether the macros `cat` and `dog` were defined to equal the same thing; and this:

```
ifelse(jim,is God,`jim is God today',
jim,is Devil,`jim is Devil today',
`jim is mortal today')#
```

would print one of three messages, depending on what I'd defined myself to be.

One more example:

```
ifelse(jim,,`jim is nothing today',
fox,hound,`the fox and the hound are
one')#
```

would print out `jim is nothing today` if `jim` were defined to be the null string, or `the fox and the hound are one` if the macros `fox` and `hound` were equal, or nothing if neither of the above were true.

Since the macros `define` and `ifelse` take arguments, it would be reasonable to expect that other macros can use arguments as well. Implicitly, all macros have arguments, but if no arguments are given in the invocation, *m4* fills the argument list of the macro with null strings. Since none of the macros `jim`, `cat`, etc. use arguments, there is no difference (except typing and processor cycles) between:

```
jim(this,is,an,argument,list)#
```

and:

```
jim#
```

Macros can use arguments, using the ubiquitous UN*X convention of `$1` to stand for the first argument, `$2` for the second, all the way up to `$9` for the ninth (`$0` stands for the macro name, and `$10` would be the first argument followed by the character 0). The arguments must be enclosed in parentheses and separated by commas. The opening parenthesis that introduces the arguments must not be separated from the macro name. For example, if *m4* were to process:

```
define(`jim',`ifelse($1,$2,`you passed $0
the same arguments',`they differ')')#
```

and if the next line was:

*The convention, described in the accompanying article, of using the '#' character to indicate hard new lines is applied in this source code.*

**masmmac.i**

```
dnl masmmac.i#
define(`tab',`ifelse(eval(len($1)<$2),1,`    ')')dnl#
define(`masm_start',`    TITLE $1#
      .386p#
divert(1)dnl#
EXTRN _next:NEAR,_docol:NEAR`'declare(_next)declare(_docol)#
divert(2)dnl#
DATA SEGMENT USE32 BYTE PUBLIC#
ASSUME ES: DATA#
__filename   db len($1),"$1",0#
divert(3)dnl#
DATA ENDS#
HEADS SEGMENT USE32 DWORD PUBLIC#
ASSUME FS: HEADS#
divert(4)dnl#
HEADS ENDS#
STACK SEGMENT USE32 BYTE STACK#
ASSUME SS: STACK#
divert(5)dnl#
STACK ENDS#
DICT SEGMENT USE32 BYTE PUBLIC#
ASSUME CS: DICT, DS: DICT#
divert(6)dnl#
DICT ENDS#
END')dnl#
define(`declare',`define(`$1_declared',1)')dnl#
define(`name',`ifelse($2,,$1,$2)')dnl#
define(`qname',`ifelse($2,,qt()$1`'qt(),qt()$2`'qt())')dnl#
define(`qt',")dnl#
define(`qt1',`define(`qt',")')dnl#
changequote(<,>)dnl#
define(<qt2>,<define(<qt>,')>)dnl#
changequote(`,')dnl#
define(`attrib',1)dnl#
define(`set_immediate',`define(`attrib',`eval(attrib|2)')')dnl#
define(`unset_immediate',`define(`attrib',`eval(attrib&~2)')')dnl#
define(`xyzzylast',0)dnl#
define(`last',`xyzzylast`'define(`xyzzylast',`__hd_$1')')dnl#
define(`new_wordlist',`define(`xyzzylast',0)')dnl#
define(`make_word',`define(`xyzzycurrent',$1)divert(1)dnl#
PUBLIC __hd_$1,__cf_$1`'declare(__hd_$1)declare(__cf_$1)#
divert(2)dnl#
__nm_$1      tab($1,3)db len(name($1,$2)),qname($1,$2),0#
divert(3)dnl#
__hd_$1      tab($1,3)dd last($1),__nm_$1,attrib(),__cf_$1
             dd __filename,lineno(),0,0
divert(5)dnl#
__cf_$1      tab($1,3)dd runtime()#
dnl')dnl#
define(`ref1',`ifelse(__cf_$1_declared,1,,divert(1)dnl#
EXTRN __cf_$1:NEAR`'declare($1))#
divert(5)dnl#
             dd __cf_$1#
dnl')dnl#
```

*(masmmac.i listing continues on next page.)*

```
jim(FORTH,love)#
```

and FORTH was defined to be love, the result would be:

```
you passed jim the
same arguments#
```

If the you made the mistake of typing:

```
jim (FORTH,love)#
```

and FORTH was defined to be love, the output would end up being:

```
you passed jim the
same arguments
(love,love)#
```

(The latter example would operate like this because *m4* will only process an argument list as an argument list if the macro name is immediately followed by an opening parenthesis. This is a result of the function that parses the input stream. It can return any of several values, and the value returned for a string that could be a macro name differs from the value that is returned for a string that could be a macro name followed by an opening parenthesis.)

Some of the other useful macros are `pushdef` (used like define, pushes the old definition of a macro onto a stack), `popdef` (pops the definition back), `changequote` (changes the quote characters from the default ` and '), `eval` (evaluates its argument as an arithmetic expression), `include` (begins reading from another named file), `divert` (stores the output in a temporary file), `undivert` (prints the contents of the diverted temporary file), `dnl` (deletes all the text following it up to and including the next new line), `include` (begins reading from a named file), `sinclude` (also reads from a named file, but silently ignores missing files), `len` (returns the length of its argument), `substr` (returns a substring of its first

argument), and index (returns the position of its second argument in its first).

The macros divert and undivert are complementary. They expect a numeric argument, which corresponds to the diversion number. If divert is passed a number outside the range of zero to nine, inclusive, the output is thrown away (which is good for getting rid of lots of new lines if you're defining several macros at once). undivert will bring in the diversion that corresponds to its argument. If the diversion doesn't exist, or undivert is invoked without arguments, the command is ignored. Diverting to zero will actually output to the standard output. If undivert is passed several arguments, it will undivert the output in the order specified (for example, undivert(3,4,2) will bring in diversion 3, then 4, then 2). If undivert is not called, all diverted output will be added to the standard output when *m4* runs out of input. undivert will throw away the text it is bringing in (i.e., you can't undivert the same diversion twice). The value of undivert (i.e., the string it returns) is the null string, not the diversion it is outputing. Thus, this:

```
define(`mystuff',undivert(2))#
```

would define mystuff to be the null string, not whatever was in diversion 2. In all cases, the diverted text is not rescanned for macros when it is brought in. The macro divnum contains the number of the current diversion.

The macro substr is called with three arguments. The first is the string that is to be chopped up, the second is the index into the string of the substring (zero based), and the third is the length of the substring. If the length is longer than the remainder of the string, or the length isn't given, the substring will be the end of the string starting at the index. For example, this:

```
substr(`The quick brown fox jumped
over the lazy dog',4,5)#
```

would print

```
quick#
```

The index macro expects two arguments. The first is the string to be matched, and the second is the string to be found. For example, this:

```
changequote(<,>)#
index(<Why, oh why, can't he talk
about FORTH?!?>,<oh why>)#
```

would print:

```
#
5#
```

The reason I used the changequote macro is that I wanted to embed a single quote in the first argument to index. It's generally a good idea to change the default quote characters if you are going to use a string with an embedded quote. This will prevent *m4* from doing strange things with the arguments. Also, commas embedded in a quoted string are protected. Since a quoted string is taken by *m4* to be a single word, the commas in the string are not used as argument separators.

If you want to separate a word from a macro name, putting `` ' between them tells *m4* to treat them as two separate words. For example, this:

```
truename(name1,name2)qt()#
```

(taken from the source code with this article, truename returns either its second argument if it is non-null, or its first argument; and qt returns the default quote character) will print:

```
name2qt()#
```

```
define(`ref2',`divert(5)        dd
__cf_`'xyzzycurrent`'dnl`'')dnl#
define(`ref',`ifelse($1,self,`ref2()',`ref1($1)')')dnl#
define(`refs',`ref($1)#
ifelse($2,,,`ref($2)')`'dnl#
ifelse($3,,,`ref($3)')`'dnl#
ifelse($4,,,`ref($4)')`'dnl#
ifelse($5,,,`ref($5)')`'dnl#
ifelse($6,,,`ref($6)')`'dnl#
ifelse($7,,,`ref($7)')`'dnl#
ifelse($8,,,`ref($8)')`'dnl#
ifelse($9,,,`ref($9)')`'dnl')dnl#
define(`xyzzyruntime',`_docol')dnl#
define(`runtime',`ifelse(xyzzyruntime,self,__pf_`'xyzzycurrent,xyzzyruntime)')dnl#
define(`set_runtime',`define(`xyzzyruntime',$1)')dnl#
define(`lineno',0)dnl#
define(`set_line',`define(`lineno',$1)')dnl#
define(`literal',`ref(lit)#
            dd $1#
dnl')dnl#
```

**dot.m**

```
include(masmmac.i)#
masm_start(dot.m)#
make_word(dot,.)#
ref(stod)#
literal(0)#
refs(ddotr,semis)#
```

**dot.s**

```
#
        TITLE dot.m#
        .386p#
EXTRN _next:NEAR,_docol:NEAR#
PUBLIC __hd_dot,__cf_dot#
EXTRN __cf_stod:NEAR#
EXTRN __cf_lit:NEAR#
EXTRN __cf_ddotr:NEAR#
EXTRN __cf_semis:NEAR#
DATA SEGMENT USE32 BYTE PUBLIC#
ASSUME ES: DATA#
__filename   db 5,"dot.m",0#
__nm_dot     db 1,".",0#
DATA ENDS#
HEADS SEGMENT USE32 DWORD PUBLIC#
ASSUME FS: HEADS#
__hd_dot     dd 0,__nm_dot,1,__cf_dot#
             dd __filename,0,0,0#
HEADS ENDS#
STACK SEGMENT USE32 BYTE STACK#
ASSUME SS: STACK#
STACK ENDS#
DICT SEGMENT USE32 BYTE PUBLIC#
ASSUME CS: DICT, DS: DICT#
__cf_dot     dd _docol#
             dd __cf_stod#
             dd __cf_lit#
             dd 0#
             dd __cf_ddotr#
             dd __cf_semis#
DICT ENDS#
END#
```

Since I would actually prefer the output to be (assuming qt was defined to be "):

```
name2"#
```

I would have to use the line:

```
truename(name1,name2)`'qt()
```

which does the job. Notice that the parentheses are not strictly necessary after the invocation of qt, but I leave them in anyway to remind me that it's actually a macro.

It should be obvious from some of my examples and the source code with this article that I primarily use *m4* to generate assembler source code. Although the examples have been fairly trivial, it is possible to build a very powerful macro preprocessor that will transform near English statements into the confusing spaghetti that an assembler expects. The *m4* source code that accompanies this article includes the macro package that I wrote for that purpose. The file `masmmac.i` contains the definitions of the macros that turn `dot.m` into `dot.s`. The last two files are the *m4* and the assembly language source, repectively, for this:

```
: . S>D 0 D.R ;
```

Notice I did this without using anything I didn't describe earlier. I'm still trying to tune the implementation but it is, for the most part, exactly what I'll use in the core Forth system.

Because *m4* writes its output to the standard output device, to use it as a preprocessor you must use a UN*X facility called I/O redirection. In any of the shells, the character > means that the standard output of the previous command is to be put into a file. Thus, this built-in dependency line from a make file would use *m4* to process macro files and put the results into an assembly language source file:

```
.m.s:
        m4 $< > $@
```

which would tell *make* that if it found a line like this:

```
foo.s: foo.m
```

it should pass this to *sh* :

```
        m4 foo.m > foo.s
```

At this point, I'm beginning to realize that maybe I was a bit too ambitious to tackle all the UN*X utilities I'll be using on this project in one article. In my next article, I'll discuss the complementary utilities *lex* and *yacc*. In the meantime, if you want to learn more about the things I discussed in this article (and believe me, there's a lot more to learn), you can take a trip either to your local public library, or to your local computer-oriented bookstore. This article barely scratches the surface of the UN*X utilities I've briefly outlined. A definitive treatment of any of these utilities could take a book. The features I've discussed, however, form a large enough subset to perform some useful work. Since I'm going to use these features (and only these features) in later articles, I felt it would be appropriate to introduce them now.

Although access to a UN*X system is not 100% necessary to understand this article, you will probably find it extremely helpful. You can get access to a UN*X system for less than you probably think. If you live in California, Netcom Online provides personal dial-up accounts for $19.50 a month (or $17.50 if you pay by credit card), which provides you with a home directory, access to several hundred UN*X utilities, e-mail services, USENET news, and hundreds of gigabytes of freely available source code. Although Netcom is based in San Jose, they have local dial-up lines in most areas of California, and one in Oregon and one in Washington. If you don't live in California, get in touch with the computer department of your local university. They should be able to steer you to a UN*X service provider.

---

The introductory article describing the FSAT project, which aims to provide the advantages of both Forth and UN*X, appears in the last issue. The author's e-mail address is `jim@netcom.com`. He'd appreciate any comments you may care to make about the project, and will reply to all messages sent to that address, provided they pertain to technical aspects of the project, and not motivational aspects.

# INTRAN—an Integer Formula Translator

*J.V. Noble*
*Charlottesville, Virginia*

*INTRAN is a simple, recursive, integer expression parser that permits formulas to be embedded in Forth words. The parser directly represents the Backus-Naur statement of its grammar as (recursive) Forth code. The resulting compiled code adds about 1Kb to the Forth kernel.*

In a letter to *Forth Dimensions* (XIV/3), Peter Roeser presented a wish list for Forths of the future. Some wishes would require consensus from the Forth community, while others (like the C interface) are already available. One wish—that Forth be broadened to encompass arithmetic expressions (including parentheses) á la Fortran, Pascal, BASIC, or C—is within the capacity of individual Forthniks (such as the author) to grant. This note presents an integer expression translator—INTRAN—in a little over a kilobyte of code.

**To make porting, maintaining, and modifying easier, I tried to keep definitions dialect-invariant.**

INTRAN came about because I needed to translate integer expressions appearing as indices in Fortran DO loops. That is, Fortran permits the construction

```
    DO 100 I=(J-K)/2, (J+K)/2+M, K/4
    ...stuff...
100 CONTINUE
```

The formula translator described in my book *Scientific Forth: a Modern Language for Scientific Computing* (available from the Forth Interest Group) parses mixed (single- and double-precision, real and complex) floating-point expressions. Unfortunately, for technical reasons (having to do with how I implemented operator overloading) the formula translator was unsuited to simple integer expressions. This implied a distinct parser for integer expressions. The one I wrote compiles to 702 bytes and surely could be squeezed further. Terseness, however, subverts clarity, so here I present a lengthier but more pedagogical version.

Since INTRAN is intended to be production-quality code, we must address the following programming issues:

- Simple user interface
- Ease of porting, maintenance, and extension
- Bulletproof error handling

And since this note is intended to illustrate general principles, I have striven to make INTRAN both clear and well documented.

We begin with the user interface: what we want INTRAN to do. Primarily, we want INTRAN to embed infix expressions into word definitions. Just as HS/FORTH, e.g., allows assembler to coexist with high-level Forth through locutions like

```
: HI-LEVEL
  words ... [%" assembler words " %] ...words...
;
```

we would like to insert formulae into words, with their translations compiled immediately as Forth. My scientific formula translator uses F"..." as in

```
: FORMULA
  F" a = (b+c)/(c+d)"   ;
```

to do precisely this. The notation has proven so convenient and transparent that I chose a similar one for INTRAN. If we define a word

```
: EXPR1
  i" (I+J)/2 + K"  ;
```

(I+J)/2 + K should be translated and compiled. When we decompile EXPR1—in HS/FORTH, e.g., using the word SHOW—we will see

```
SHOW EXPR1      I J + 2 /  K +  ;  ok.
```

Experience shows the value of dry runs: during testing we might want INTRAN to translate interactively, without compiling. That is, when we type

```
i" (I+J)/2+K"
```

INTRAN should send the Forth translation

```
I J + 2 /  K +  ok
```

to the screen. To make all these things happen, i" must be state-smart. These abilities alone would make INTRAN the

# HARVARD SOFTWORKS

*NUMBER ONE IN FORTH INNOVATION*

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

You already know HS/FORTH gives more speed, power, flexibility and functionality than any other implementation. After all, the majority of the past several years of articles in Forth Dimensions has been on features first developed in HS/FORTH, and many major applications discussed had to be converted to HS/FORTH after their original dialects ran out of steam. Even public domain versions are adopting HS/FORTH like architectures. Isn't it time you tapped into the source as well? Why wait for second hand versions when the original inspiration is more complete and available sooner.

Of course, what you really want to hear about is our **SUMMER SALE!** Thru August 31 only, you can dive into Professional Level for $249. or Production Level for only $299. Also, for each utility purchased, you may select one of equal or lesser cost free.

Naturally, these versions include some recent improvements. Now you can run lots of copies of HS/FORTH from **Microsoft Windows** in text and/or graphics windows with various icons and pif files available for each. Talk about THE tool for hacking Windows! But, face it, what I really like is cranking up the font size so I can still see the characters no matter how late it is. Now that's useful. Of course, you can run bigger, faster programs under DOS just as before. Actually, there is no limit to program size in either case since large programs simply grow into additional segments or even out onto disk.

Good news, we've redone our **DOCUMENTATION!** The big new fonts look really nice and the reorganization, along with some much improved explanations, makes all that functionality so much easier to find. Thanks to excellent documentation, all this awesome power is now relatively easy to learn and to use.

And the Tools & Toys disk includes a complete mouse interface and very flexible menu support in both text and graphics modes. **Update to Revision 5.0**, including new documentation, from all 4.xx revisions is $99. and from older systems, $149. The Tools&Toys update is $15. (shipping $5.US, $10.Canada, $22.foreign)

**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

**Figure One.** Rules for INTRAN's grammar.

```
<assignment>   ->   <id> = <expression>
<expression>   ->   <term> | <term> & <expression>
<term>         ->   <factor> | <factor> % <term>
<factor>       ->   ( <expression> ) | <id> | <literal>
<literal>      ->   digit {digit}*
<id>           ->   letter {letter | digit}*  {ε | @}
```

answer to Mr. Roeser's prayers.

However, experimenting with early versions of INTRAN has proven the worth of several other features. Sometimes I need expression fragments, like (I+J)/2+K, that leave a result on the stack, but other times I wish to embed formulas in assignment form,

```
B = (I+J) / 2 + K
```

that translate to something like

```
I J + 2 / K + B !
```

Fortunately it is easy to do both with little extra code.

Here is another problem of interface design: what do the symbols I, J, K, and B represent? Many Forths define words I, J, and K that fetch the current values of indices (from nested loops) to the stack. For these we must omit explicit fetches (@'s). However, our expressions might well incorporate Forth VARIABLEs. But a VARIABLE must be followed by @ for the translation to work correctly.

What about formulas including CONSTANTs or VARs (multiple code-field words—QUANs in MMSForth and VALs in other dialects): in either case @ would be incorrect. While Forth makes it possible to determine, during compilation, the type of a data structure (e.g., we could look up its CFA), the simplest course is to require the programmer to keep track of VARIABLEs. That is, at all costs we should avoid decisions and keep the program simple, especially since good Forth style tends to eschew variables (that is, formulas including variables will be rare). The programmer will indicate a VARIABLE with a @ at the end of its name; this @ will be translated (with appropriate spaces) to make the resulting Forth correct:

```
: EXPR1
  i" (I0@+J0@)/2 + K1"  ; ok

SHOW EXPR1
I0 @ J0 @ + 2 /  K1 +  ; ok
```

In an assignment like B=(I+J)/2+K we need to know what B refers to. A VARIABLE is stored to by the phrase B ! while a VAR uses the locution IS B. Although good programming practice frowns on it, we can even modify CONSTANTs with the phrase ' B !. Since the latter phrase works equally well for CONSTANTs, VARIABLEs, and VARs, it solves our problem and requires no decisions (only in HS/FORTH—will not work in other Forths).

Finally, we want the parser to report bad input, and to ignore white space inserted for readability. With these provisos, we have completely specified the user interface.

To make porting, maintaining, and modifying easier, I have tried to keep definitions dialect-invariant, to name the key actions telegraphically, to comment thoroughly, and to make the program structure as transparent as possible. In particular, the self-referential nature of the grammatical rules embodied in a formula translator can be most easily and clearly expressed through recursion.

The context-free grammar for INTRAN (see *Scientific Forth* and references contained therein) can be expressed as rules (see Figure One).

The notation for these rules contains some shorthand: & stands for + or -, % for * or /, and | means "or". A superscript * means "zero or more". The Greek letter ε means "void", hence {ε | @} stands for "nothing or the symbol @".

The preceding grammar lends itself naturally to a recursive programming style. For the pseudo-Forth (also in reverse order of definition!), see Figure Two.

To implement these ideas as working code we need to define words that can recognize +, -, *, /, and =, as well as parenthesized expression fragments and id's. A further restriction on the words that find numeric operators is that they must never be enclosed within parentheses. That is, we want * in (a+b)*(c-d) to be found before + or -.

Then we must choose how to represent the input string (and the substrings obtained as we decompose it into terms, factors, and parenthesized expressions), as well as the operators corresponding to each step in the decomposition.

To minimize crashes during development, we begin with the purely interpretive function of i". When that works properly we can worry about its compiling function. As usual, we begin with the low-level words. We need a word to find either of a pair of characters. Since we need to do this with + and -, * and /, and also = and = (this last so we can use the same word to find =), there are enough cases to justify the memory overhead of a defining word

```
: a|b
  CREATE  D,
  DOES>  D@  ROT  UNDER
         =  -ROT  =  OR  ;

used as

ASCII +  ASCII -  a|b  +|-
ASCII *  ASCII /  a|b  *|/
ASCII =  ASCII =  a|b  =|=
```

At run time, the words +|-, *|/, and =|= compare the TOS (top of stack) with two built-in numbers; if either matches, the words return *true* (-1). The plan is to write a generic character-finding routine that will search a string byte by byte, and with each byte EXECUTE one of the above words. That is, the search word will take as input the location of the string and the CFA (code-field address) of the word

**Figure Two.** INTRAN's BNF grammar expressed as psuedo-Forth.

```
: <assignment>      find "="   found?
                    if   push { id   "!" }
                         push { null " ' " }
                         push { expression  BL }
                         <expression>  print  print
                    else push { expression  BL }
                         <expression>
                    then  ;

: <expression>      find "&"   found?
                    if     rearrange   <expression>
                    else   arrange
                    then   <term>  ;

: <term>            find "%"   found?
                    if     rearrange   <term>
                    else   arrange
                    then   <factor>  ;

: <factor>     literal or id?  if  print  exit  then
               (<expr>)?
               if    remove()  <expression>
               else  error.mss  then  ;
```

corresponding to the particular character(s) being sought.

The generic character-finding routine mentioned above is `find2`. The first moderately complex word in INTRAN, it embodies a not altogether necessary trick, a sort of "hack." The problem is this: we need to go through the input string one byte at a time, looking for either of two characters representing operators. The CFA of the word that tests the input is one argument, the others being the beginning and end of the string. Only "exposed" operators—those not hidden within parentheses—can be "found." So `find2` must also check that the parenthesis-level is also zero (recall the old trick for balancing parentheses in long Fortran expressions: counting from the left, start at zero and add one for each "(", subtract 1 for each ")"; if the last ")" coincides with a zero count, the parentheses are balanced).

All four variables—parenthesis count, string pointers, and CFA—are temporary, so we are tempted to keep them on the stack while `find2` searches, and to drop them thereafter. By using a `DO...LOOP` with the string pointers as limits, we automatically put two of the arguments on the rstack. This leaves only the parenthesis count and CFA on the stack; however, as the `DO` loop executes, both must be accessed. While not impossible to an accomplished stack gymnast, the necessary manipulations tend to hide what is going on. They are also prone to error. Worse, the search must generate a "semi-flag": the address of a found operator, or a zero to indicate failure. This is tricky to program—one must `LEAVE` the loop if the operator is found, but it is then hard to determine whether or not the loop terminated early.

Perhaps `DO...LOOP` is not the appropriate looping mechanism. So I tried `BEGIN...WHILE...REPEAT`. Now one may comfortably put the temporary variables on the rstack (difficult in a `DO...LOOP`, where the limits must be accessible on the rstack), but the termination condition(s) are no longer automatic. Endless loops are easy to produce inadvertantly.

What about looping by recursion? This is even more horrible because the CFA, needed at each iteration, would be buried if placed on the rstack, but would be relatively inaccessible on the parameter stack as well. It would have to be off-loaded to a `VARIABLE`, which is what we were trying to avoid. So in the end, compromise: I returned to `DO...LOOP` and off-loaded the parenthesis count to its own named variable, `() level`.

But I began this section by advertising a cute trick in `find2`. Where is it? In order to leave the result ( -- adr\|0 ), I first copy the CFA (of the comparison routine x\|y) on the rstack with `DUP>R`, then put it below the loop limits using `-ROT`. `DO` now moves the limits to the rstack and begins executing. The CFA is on the TOS. `IF` an exposed operator is found, we `LEAVE` the loop early, replacing the CFA by the current index `I`. Finally, we retrieve the saved CFA and compare it to TOS with the phrase `DUP R> <>`. If the loop terminated normally, TOS is still the CFA and the comparison yields *false*. Whereas, when the loop terminates early (because it has found what it was looking for), the TOS is the desired address, which cannot be the CFA, so the comparison yields *true*. The final ANDing together of flag and address then leaves the desired semi-flag.

Another look at the INTRAN listing reveals that, after each word was tested, I followed it with a comment line "`\ OK time date`". This discipline helped me to be both thorough and systematic in the endless quest for bug-free code. An editor with a time/date stamp lessens the tedium of this necessary chore.

The next major issue is how to represent the input string, as well as the substrings we are going to find as we decompose it. The first time I ever wrote a formula translator, I actually defined a stack to hold strings, and placed each string and substring in a separate stack location, with the operators on a parallel stack. Thus a formula such as A=B+C would lead to

| *Stage 0* | | *Stage 1* | | *Stage 2* | |
|---|---|---|---|---|---|
| A=B+C | nop | A | ! | A | ! |
| | | B+C | nop | C | + |
| | | | | B | nop |

Some sort of stack is manifestly demanded by recursion. Eventually I realized a string stack was excessive: since each fragment appears once and only once, it is enough to store pointers marking the beginning and end of each fragment, rather than copying the fragments themselves. We still need a stack to pass the arguments during recursive calls, however, so my next try was a stack three cells wide, to hold the pointers and a token for the operator. For technical reasons, this is still the method I use in my scientific formula translator. But several attempts to write INTRAN made clear that the parameter stack would suffice to store the pointers and operator. So the words <expression>, <term>, and <factor> will expect the input stack picture

```
( beg end .op )
```

where .op stands for an operator token, represented as the ASCII codes 32d, 42d, 43d, 45d, and 47d (BL, *, +, -, and /).

The words to determine whether a string is an identifier or a literal integer are straightforward. They could as well have been programmed as state machines, which would have made it simple to enforce length rules (i.e., how long valid numbers or ID's can be). I chose not to include such elaborations because my Forth allows ID's up to 32 characters long, and because I am unlikely to write an excessively large literal integer.

It is now time to implement the Backus-Naur grammatical rules in real, rather than pseudo-, code. Consider <expression>, whose rule is

```
<expression> -> <term> | <term> "&" <expression>
```

and whose code is

```
: <expression>   ( beg end .op -- )  -bl-
   >R DDUP  CFA' +|- find2       \ find&
   R>  OVER                      \ found?
   IF    rearrange    RECURSE
   ELSE  PLUCK   THEN   <term>  ;
\ OK 17:36:42  3/2/1993
```

With the exception of the word -bl- (discussed below), I have translated the rule directly into Forth. One crucial phrase is

```
CFA' +|- find2
```

which locates an exposed + or -. The other is the IF...ELSE...THEN that executes <expression> <term> when there is an exposed conjunction, but only <term> when there is none. The words >R, R>, rearrange, DDUP, and PLUCK are mere "glue" that do not express the algorithm but are nonetheless necessary to its actual performance.

Of course recursion is not absolutely necessary: by theorem, recursion can always be removed from a recursive program (sadly, the theorem does not give specifics). However, recursion offers a decisive advantage over non-recursive indefinite loops. The latter demand an explicit stopping condition such as "test the stack depth to see if there are no more arguments"; whereas recursion implicitly keeps track of execution. To guarantee this with a multiply recursive algorithm such as INTRAN, we must ensure, first, that each parsing word takes the same number of arguments; and second, that each leaves nothing on the stack. Thus, to make a recursive call to <expression> followed by a call to <term>, we must put *two* sets of arguments on the stack which the two calls will consume. This is precisely what rearrange does. On the other hand, the second branch (that only calls <term>) requires but one set of arguments, which PLUCK takes care of.

Note that <factor> has to call <expression> before the latter has been defined. Some dialects permit forward vectoring using a word like DEFER, and in fact HS/FORTH offers several forward-vectoring methods. But all Forths permit the simple vectoring method I used, namely to define a VARIABLE to hold the code-field address of the word to be executed. The phrase 'expression @ EXECUTE in <factor> will execute whatever word has its CFA stored in 'expression.

Finally, executing the phrase

```
CFA' <expression>   'expression !
```

immediately after defining <expression> fulfills the forward reference. This reference achieves indirect recursion (<expression> calls <term> which calls <factor> which calls <expression>) in addition to the direct recursion found in <expression> and <term>.

With all the advantages of brevity, directness, and simplicity that recursion brings to a program like INTRAN, one wonders whether there are countervailing disadvantages. The chief one is the ever-present danger of mistreating the stack in such a way as to produce an endless loop that overwrites vital things. Debugging tools like HS/FORTH's TRACE and SSTRACE are vital to making sure recursive definitions behave themselves.

The reader will find the word -bl- sprinkled through the code. What does it do, and why is it there? I wanted to allow formulas to contain optional white space for clarity. Unfortunately, there does not seem to be any simple method, short of redefining it completely, to set my Forth's version of WORD to ignore blanks. (WORD is the key component of TEXT, that reads input terminated with a given character, to the scratchpad (PAD).) Since using TEXT was easier than redefining it from scratch, I was left with a string (possibly) full of blanks.

How to rid the input of blanks? *A priori,* it seems most efficient to de-blank the entire input string before parsing. The (two) words in Figure Three accomplish this:

The algorithm is simple: having first used the system word -TRAILING to eliminate trailing blanks, search (from the left, rightward) for the first blank. Save the address where this occurs. Then continue rightward to the first non-blank character. Compute how many characters are in the tail of the string. Slide the tail leftward to the first blank, and adjust the end-of-string pointer. Repeat until all blanks have been eliminated.

Oddly, the above procedure is neither so compact nor so

**Figure Three.** "De-blanking" the input string before parsing.

```
: skip   ( end beg char cfa -- end beg' )  D>R
         1- BEGIN 1+  DDUP =   OVER C@   DR@ EXECUTE  OR  UNTIL  DRDROP  ;

: -BL  ( end beg -- end' beg)                  \ strip blanks out of a string
       UNDER  - 1+  -TRAILING                  \ strip trailing blanks
       OVER + 1-  SWAP                         ( end" beg)
       DUP>R  BEGIN   BL CFA' =   skip         \ -> 1st BL
                      DDUP                     ( end adr end adr)
                      BL CFA' <>  skip         \ -> non-blank
                      PLUCK                    ( end adr adr+n-1)
                      DDUP  <                  \ not at the end yet
              WHILE   OVER  DDUP  -                  ( end adr src dst #bl)
                      -ROT  D>R  -ROT                ( #bl $end adr)
                      DDUP - 1+  DR>  ROT  CMOVE     ( #bl $end adr)
                      -ROT  SWAP  -  SWAP
              REPEAT  DDROP  R>  ;
```

**Figure Four.** Moving the pointers.

```
: -bl-      ( beg end .op -- beg' end' .op)
       >R     1+  BEGIN  1-  DUPC@  BL <>  UNTIL \ skip leading
       SWAP   1-  BEGIN  1+  DUPC@  BL <>  UNTIL \ skip trailing
       SWAP   R>  ;
```

efficient as my "afterthought" method (when I realized blanks should be allowed). Whereas the preceding code adds 135 bytes, the "afterthought" adds 57. How does it work? We decompose as though the blanks were not present. The substring pointers point to the ends of (sub)strings with (possibly) leading and trailing blanks. It is much easier to advance the beginning pointer rightward past the leading blanks and the end pointer leftward past the trailing ones, than to actually compose a new, blankless string from one with blanks sprinkled through it. So we compose a word that moves the pointers as noted (see Figure Four), and then apply it whenever a new substring has been dissected out and its pointers placed on the stack. The definition `-bl-` (the name suggests that blanks are removed at both ends) requires 47 bytes (in an indirect-threaded system), and the five subsequent references to it add another ten bytes.

We are nearing the end of the exposition. The final definition (the initial one, were we so foolish as to design top-down) is `i"`. Before we add the compiling abilities described above, we only require that it acquire the text, place pointers to the ends of the string, as well as a do-nothing operator token, on the stack, and invoke the first parsing word, `<assign>`:

```
: i"    ASCII " TEXT     ()_ok
  \ check for balanced parens
        PAD  adr>ends   BL  CR  <assign>
;
OK 17:27:16  3/2/1993
```

We check for balanced parentheses before starting to parse, since this obviates multiple tests of the parenthesis level during execution.

Having tested the word `i"` on many different cases and having tried (unsuccessfully, one hopes!) to make it fail, we are now ready to try out the most dangerous part of the development: granting `i"` the power to compile expressions. Many Forths (and the new ANS standard) include a word like EVAL that will compile directly from a string. If your Forth can do this, all that is needed is to redirect the output from the screen to a buffer, convert that to a counted string, and EVALuate the string. HS/FORTH has an equivalent form of redirection, namely the ability to load from a buffer (the text in the buffer must be terminated by double-0). The (non-standard) word that does this is `MLOAD` which expects a segment descriptor and offset on the stack. The definitions needed to implement expression compilation are then as shown in Figure Five *(on next page)*. The only major changes from the non-compiling version of `i"` are the state-dependent `IF...ELSE...THEN` and making `i" IMMEDIATE`, so it can do its work within a word being defined.

*(Code begins on next page.)*

Julian V. Noble is a professor of physics at the University of Virginia and the author of *Scientific Forth*. He earned his B.S. at CalTech in 1962 and his Ph.D. at Princeton in 1966, and once designed a floating-point co-processor for the Jupiter Ace.

**Figure Five.** Implementing expression compilation.

```
256 SEGMENT intran      \ make a named segment to hold output from i"  "

: i"->MEM     intran @  0  256 0 FILLL          \ initialize buffer
              intran  OPEN-MEM  >MEM   ;        \ output -> buffer

: MEM->     intran @  0  MLOAD                  \ load from intran
           CLOSE-MEM  ;

: i"     ASCII " TEXT     ()_ok                    \ input, check ()
         PAD  adr>ends    BL  CR                   \ set stack
         STATE @                                   \ compiling?
         IF  i"->MEM                               \ vector to intran
             <assign>                              \ output the FORTH code
             CRT                                   \ return output to CRT
             MEM->                                 \ load from intran
         ELSE  <assign>   THEN   ;   IMMEDIATE     \ otherwise -> display
\ OK 12:21:28  3/3/1993
```

**INTRAN code listing.**

```
\ Mini expression parser
\ version of 11:50:06  3/6/1993
\ compiles to 1239 + 256 bytes
TASK INTRAN

\ This program was written to be used with HS/FORTH, an indirect-
\ threaded FORTH for the PC. It contains several non-standard words
\ and usages (especially not ANS !) but has been designed to be easy
\ to translate to your own dialect.

: a|b   ( c1 c2 --)   CREATE  D,
        DOES>  D@  ROT  UNDER  =  -ROT  =  OR  ;  ( c -- f)
\ Defining word. Child words contain 2 built-in characters.
\ Test char on stack and return true if it is either built-in char.
\ OK 20:31:59  3/1/1993

ASCII +  ASCII -  a|b  +|-
ASCII *  ASCII /  a|b  *|/
\ OK 22:10:12  3/1/1993

VARIABLE ()level    0 ()level !     \ holds current parens level

: inc()    ( c --)   ASCII )  OVER  =   SWAP   ASCII ( =   -   ()level +!  ;
\ increment/decrement ()level; ( increments, ) decrements.
\ OK 20:44:29  3/1/1993

: adr>ends  ( $adr -- beg end)  COUNT  OVER +  1-  ;
\ convert address of counted string to pointers to beg and end of text
\ OK 20:34:18  3/1/1993

: find2     ( beg end cfa -- adr|0 )
            ()level 0!   DUP>R                          \ initialize
            -ROT  DO  I C@   DUP    inc()               \ adjust ()level
                      OVER   EXECUTE    ( -- cfa f)     \ test input
                      ()level @  0=                     \ exposed?
```

```
                    AND                                     \ found?
                    IF  DROP  I  LEAVE   THEN               ( -- adr)
          -1 +LOOP
          R>  OVER  <>  AND  ;                              \ ( adr | 0 )
\ search text from beg to end and test using routine pointed to by cfa
\ leave character-address if found, 0 otherwise.
\ OK 22:03:34  3/1/1993

: print   ( beg end .op -- )   -ROT                 \ move op token
         DUP C@  >R                                 \ save last char on rstack
         OVER - TYPE                                \ type all but last char
         R>    DUP  ASCII @ =                       \ last char = @ ?
         IF  SPACE  THEN                            \ emit space
         EMIT   SPACE                               \ type last char
         SPACE  EMIT  SPACE  ;                      \ type operator
\ print out the string and operator on the stack
\ OK 15:17:49  3/2/1993


: WITHIN      ( k m n --f)   ROT  UNDER  MIN  -ROT  MAX  =  ;
\ return "true" if  n >= k >= m, else "false"
\ Note:  UNDER is TUCK in ANS
\ OK 21:44:15  3/2/1993

: digit?      ( c -- f)          ASCII 0  ASCII 9  WITHIN ;
: letter?     ( c -- f)   32 OR  ASCII a  ASCII z  WITHIN ;

: <id>      ( beg end -- f)      \ <id> -> letter {letter|digit}* { |@}
           DUPC@  ASCII @ =  +   \ ignore trailing @
           SWAP  DUPC@  letter?  -ROT
           1+ SWAP
           DO  I C@  DUP
               letter?   SWAP   digit?   OR   AND
           -1 +LOOP  ;
\ OK 22:37:51  3/2/1993

: <#>       ( beg end -- f)          \ <#> -> {digit}+
           1+  SWAP                  ( end+1 beg)
           -1 -ROT                   ( -1 end+1 beg)
           DO  I C@  digit?   AND    LOOP  ;
\ OK 12:55:49  3/2/1993

: simple?    ( beg end -- f)  DDUP  <#>  -ROT  <id>  OR  ;


VARIABLE BLBL   8224 BLBL !   \ "blank blank"
: NULL    BLBL   DUP 1+  ;    ( -- beg end)

\ set up forward reference for <factor>
   VARIABLE 'expression
   CFA' NEXT  'expression !    \ initialize to something harmless
\ Note: CFA' is nonstandard; it means "get execution token of next word"

: (<expr>)?   ( beg end -- f)  C@ ASCII ) =   SWAP   C@ ASCII ( =   AND  ;
\ is it an expression within parentheses?
\ OK 19:53:31  3/2/1993
```

```
: <factor>  ( beg end .op -- )   -bl-  \ <factor> -> <#> | <id> | (<expr>)
           >R
           DDUP  (<expr>)?                  \ enclosed?
           IF  1-  SWAP 1+ SWAP             \ remove ()
               R>  'expression  @ EXECUTE   \ <expression>
               EXIT
           THEN
           DDUP  simple?                    \ <id> or <literal>
           IF    R>  print
           ELSE  RDROP  CRT ." INCORRECT EXPRESSION"  ABORT  THEN  ;
\ OK 11:46:25  3/6/1993

\ auxiliary words for <term>
: -bl-      ( beg end .op -- beg' end' .op)
           >R    1+  BEGIN  1-  DUPC@  BL <>  UNTIL  \ end' <- end
           SWAP  1-  BEGIN  1+  DUPC@  BL <>  UNTIL  \ beg  -> beg'
           SWAP   R>  ;
\ strip blanks from both ends of text
\ OK 20:07:32  3/2/1993

: 3SWAP   ( a b c d e f -- d e f a b c)    6 ROLL 6 ROLL 6 ROLL  ;
\ Note: in F83 and ANS it would be 5 ROLL

: rearrange      ( beg end adr .op -- adr+1 end .op'  beg adr-1 .op )
                 >R   DUP>R                    ( -- beg end adr)
                 1+ SWAP  ROT  R@ C@           ( -- adr+1 end beg .op')
                 SWAP  R>  1-   R>   ;
\ OK 17:05:24  3/2/1993

: <term>  ( beg end .op -- )     -bl-     \ trm -> fctr | fctr % trm
          DUP>R  BL <>     \ not .nop?
          IF  BL NULL R> 3SWAP  RECURSE   print   EXIT  THEN
          \ put  NULL .op  above   stuff .nop , <term> print
          DDUP  CFA' *|/  find2             \ find%
          R>  OVER                          \ found?
          IF    rearrange    RECURSE
          ELSE  PLUCK   THEN   <factor>  ;
\ Note: PLUCK is NIP in ANS
\ OK 11:46:31  3/6/1993

: <expression>   ( beg end .op -- )  -bl-     \ expr -> term | term & expr
                 >R DDUP  CFA' +|- find2      \ find&
                 R>  OVER                     \ found?
                 IF    rearrange    RECURSE
                 ELSE  PLUCK   THEN   <term>  ;
\ Note: PLUCK is NIP in ANS
\ OK 17:36:42  3/2/1993

CFA' <expression>  'expression !   \ resolve forward reference in <factor>

: put0    ( beg end -- beg end+1)
          DUP>R  OVER -  1+  >R  DUP DUP 1+ R>  <CMOVE
          ASCII 0  OVER  C!   R>  1+  ;
\ replace text abcd... by 0abcd...  and update pointers
\ Note:   use CMOVE> in ANS
```

```
: fix-    ( beg end -- beg end f)      \ leading "-" -> leading "0-"
          OVER C@  ASCII - =           \ leading - ?
          IF  put0  THEN   ;
\ OK 11:43:23  3/3/1993

ASCII = DUP  a|b =|=                        \ to find =

: <assign>  ( beg end .op -- )   -bl-          \ eliminate spaces
      >R  DDUP   CFA' =|= find2                \ find=
      ?DUP                                     \ found?
      IF    ( -- beg end adr )
            DUP>R  1-  SWAP   ASCII !  SWAP     ( -- beg adr-1 "!" end )
            >R   NULL  ASCII '     ( -- beg adr-1 "!" beg' end' "'")
            R> R> 1+  SWAP   ( -- adr+1 end)
            R> -bl- >R                         \ eliminate spaces
            fix-                               \ "-" -> "0-"
            R>   <expression>   print   print
      ELSE  fix-   R>  <expression>   THEN   ;
\ parse as assignment statement
\ if "=" found, save <id> etc. to assign on stack
\ if no "=" is found, parse as expression
\ Note: as implemented here, <assign> lets you store to a constant or
\       a variable. (line involving ASCII ' )
\       There does not seem to be any way to modify a CONSTANT in ANS.
\       Use a VALUE instead, and rewrite so it outputs "TO <id>"
\ OK 11:43:34  3/3/1993

: ()_ok?   ()level 0!    PAD  COUNT  OVER +  SWAP
           DO   I C@  inc()  LOOP   ()level @
           IF CRT ." Unbalanced parentheses!"  ABORT  THEN  ;
\ OK 17:52:30  3/2/1993

\ : i"    ASCII " TEXT    ()_ok?
\         PAD  adr>ends   BL  CR  <assign>  ;
\ OK 17:27:16  3/2/1993

256 SEGMENT intran      \ make a segment to hold output from i"  "

: i"->MEM     intran @  0  256 0 FILLL         \ initialize buffer
              intran  OPEN-MEM  >MEM   ;       \ output -> buffer

: MEM->    intran @  0  MLOAD                  \ load from intran
           CLOSE-MEM  ;

: i"    ASCII " TEXT    ()_ok?                    \ balanced () ?
        PAD  adr>ends   BL  CR                    \ set initial stack
        STATE @                                   \ compiling?
        IF  i"->MEM                               \ vector to intran
            <assign>                              \ output the FORTH code
            CRT                                   \ return output to CRT
            MEM->                                 \ load from intran
        ELSE  <assign>   THEN   ;   IMMEDIATE
\ Note: This method of redirecting output to a memory buffer is unique
\       to HS/FORTH. In ANS you will have to output to a counted string
\       then use EVAL.
\ OK 11:48:06  3/6/1993
```

# Terminal Input and Output

C.H. Ting
San Mateo, California

*[To work through this tutorial, you will need the random-number generator presented in the last issue's lesson.]*

### Format Output Numbers

We have discussed a few of the Forth instructions which print numbers to the screen. Here is a list of these instructions:

`.` ( n – )
Print signed n followed by a space.

`U.` ( u – )
Print unsigned u followed by a space.

`D.` ( d – )
Print signed double integer d with a space.

`.R` ( n1 n2 – )
Print n1 in n2-column format.

These instructions are sufficient for most applications in which numbers are displayed on the screen to tell a programmer to know what's going on in the computer. But to display numbers to the end user, they have to be formatted in specific ways not covered by the above list of commends.

Forth provides component instructions that enable you to format numbers in any desirable fashion. We will discuss these instructions and use a few examples to illustrate how they are used to create custom number formats.

The formatting process starts with a double integer on the top of the stack. The output is a formatted string in the free memory below the text buffer pointed to by the instruction PAD. The output string is constructed backwards, one digit at a time. While constructing the string, any ASCII characters can be inserted into the string to improve readability. The building blocks for number formatting are the following:

`<#` ( -- )
Start the number-formatting process.

`#` ( d1 -- d2 )
Divide d1 by the radix number. The quotient is left on the stack as d2. The remainder is converted to a digit, which is added to the output string.

`#S` ( d – 0 0 )
Convert all significant digits and add them to the output string.

`#>` ( d – addr n )
Terminate the number string, and leave addr and the count of the string's length for `TYPE`.

`HOLD` ( char -- )
Add an ASCII character to the output string.

`SIGN` ( n – )
Add a - sign to output string if n<0.

`BASE` ( -- addr )
Memory address containing current radix.

`DECIMAL` ( -- )
Set radix to 10 for decimal conversion.

The radix number which controls the number conversion can be changed at will. This flexibility allows the Forth user to do very interesting and powerful number formatting.

### Telephone Numbers

Telephone numbers, including the area code and even an international dialing code, can be comfortably represented by double integers. To print a telephone number with the custom format, we define the word `Phone` as in Figure One.

A double integer can be entered from the keyboard by including a period anywhere in the input number string. To print the phone number shown in the comment in Figure One, type:

```
415432.1230 Phone
```

and the properly formatted string will be printed on screen.

### Time of Day

Assume that the time of day is represented by an double integer counting 1/100ths of a second since midnight. The

output format specification is HH:MM:SS.XX, where XX represents hundredths of a second. (See the code presented in Figure Two.)

Between midnight and noon, there are 4,320,000 hundredths of seconds. If we type:

```
4320000. TIME
```

we should see 12:00:00.00 printed. Do verify it.

### Angles

For navigational purposes, global positions are represented in angles of latitude and longitude, in the form of DDD:MM'SS". (Here we are using : for degrees because we don't have the proper character for degrees.) This format is only slightly different from the above time format, and can be defined as in Figure Three.

### Radix for Number Conversions

By changing the radix stored in BASE, the Forth user can freely convert numbers from one base to another, according to the needs of the moment. Programmers must often convert numbers from decimal to hexadecimal, octal, and binary, and vice versa. The number conversions can be done simply by changing the radix with the following instructions:

```
DECIMAL
: OCTAL          8 BASE !  ;
: HEX            16 BASE ! ;
: BINARY         2 BASE !  ;
: RADIX36        36 BASE ! ;
: RADIX19        19 BASE ! ;
```

Special radix bases are sometimes very useful for particular situations. For example radix 36 is very convenient when compressing alphanumerical strings to fit into tight memory spaces, because it encompasses the ten numerical digits and the 26 characters of the alphabet. My favorite radix is 19, which is useful in encoding board locations in the Chinese (or Japanese) game of Go.

Try converting numbers among different radices:

```
DECIMAL
12345 HEX .

HEX
ABCD DECIMAL U.

DECIMAL
100 BINARY .

BINARY
101010101010 DECIMAL .
```

Real programmers impress novices by carrying a H-P calculator which can convert numbers between decimal and hexadecimal. A Forth computer has this calculator built in, besides other functions.

### Message Coding

After you have entered the code from the preceding section, try this:

**Figure One.** Telephone numbers.

```
: Phone ( d -- , print the telephone number in [415] 432-1230 format )
  <#                        ( start output string )
  # # # #                   ( convert the last 4 digits )
  45 HOLD                   ( insert - sign )
  # # #                     ( convert the next 3 digits )
  32 HOLD                   ( add a space )
  41 HOLD                   ( add right parenthesis )
  # # #                     ( add area code )
  40 HOLD                   ( add left parentheses )
  #>                        ( done formatting )
  TYPE SPACE                ( print the output string )
  ;
```

**Figure Two.** The time of day.

```
: Sextal ( -- )
  6 BASE !                        ( set radix to 6 )
  ;

: :SS   ( d1 -- d2 , divide d1 by 60 and add the remaider as 2 digits )
  ( to the output string )
  #                               ( convert 1 digit in decimal )
  Sextal #                        ( convert next digit in sextal )
  DECIMAL                         ( restore readix to decimal )
  58 HOLD                         ( add : to output string )
  ;

: Time ( d -- )
  <#                              ( start output string )
  # #                             ( convert hundredth of second )
  46 HOLD                         ( add . )
  :SS                             ( convert seconds )
  :SS                             ( convert minutes )
  #S                              ( convert hours )
  #>                              ( terminate conversion )
  TYPE SPACE                      ( print results )
  ;
```

```
RADIX36
: Message
  THE. WOLVES. ARE.
  COMING. ;
DECIMAL
```

Try the following and you will see how the message is coded and decoded:

```
RADIX36
Message D. D. D. D.

DECIMAL
Message D. D. D. D.
```

Please note that the words in Message are appended with periods to force them to be converted to double integers in radix 36. By the way, the largest number representable in radix 36 is 1Z141Z3. It is, therefore, possible to represent any six-character alphanumeric string as a double integer in radix 36.

### Terminal Input and Output

The Forth user interacts with Forth through the terminal: a keyboard to enter numbers and instructions, and a screen to display results and other information. The most elementary Forth instructions controlling the terminal I/O are:

KEY ( -- char )
Accept a keystroke from the keyboard and return the corresponding ASCII code.

KEY? ( -- f )
Return a true flag if a key was pressed.

EMIT ( char -- )
Display the character whose ASCII code is on the top of stack.

TYPE ( addr n -- )
Display a string n characters long from memory location addr.

**Figure Three.** The degrees, minutes, and seconds of angles.

```
: Angle ( d -- , print angle in the DDD:MM'SS" format )
  <#                                ( start output string )
  34 HOLD                           ( add " for seconds )
  #                                 ( decimal digit for second )
  Sextal # DECIMAL                  ( sextal digit for second )
  39 HOLD                           ( add ' for minutes )
  :SS                               ( convert minutes )
  #S                                ( convert degrees )
  #>                                ( done formatting )
  TYPE SPACE                        ( print )
  ;
```

**Figure Four.** Printing the PC's character set.

```
: Printable ( n -- n , convert non-printable characters to spaces )
  DUP 14 <                          ( 7-13 are special formatting )
  IF      DUP 6 >                   ( characters not displayable )
          IF DROP 32 THEN           ( substitute a space )
  THEN
  ;

: HorizontalASCIItable( -- )
  CR CR CR
  5 SPACES
  16 0 DO I 4 .R LOOP           ( show sequential column header )
  CR
  16 0 DO                           ( do 16 rows )
          CR I 16 * 5 .R            ( print row header )
          16 0 DO                   ( print 16 characters in a row )
                  3 SPACES
                  J 16 * I +        ( current character value )
                  Printable
                  EMIT               ( print it )
          LOOP                      ( loop for next character )
  LOOP                              ( loop for next row )
  CR
  ;

: VerticalASCIItable( -- )
  CR CR CR
  5 SPACES
  16 0 DO I 16 * 4 .R LOOP          ( show column headers )
  CR
  16 0 DO                           ( do 16 rows )
          CR I 5 .R                 ( print row header )
          256 0 DO                  ( do 16 columns )
                  3 SPACES
                  J I +             ( current character )
                  Printable EMIT
          16 +LOOP              ( skip 15 characters between columns)
  LOOP
  CR
  ;
```

**Figure Five.** The love letter.

```
VARIABLE NAME 12 ALLOT
VARIABLE EYES 10 ALLOT
VARIABLE ME   12 ALLOT

: ENTER ( addr n -- , accept a string up to n characters to the )
  ( memory area starting at addr .)
  2DUP BLANK                ( clear the memory area to spaces )
  EXPECT                    ( wait for a string up to n )
  ;                         ( characters or a return )

: VITALS ( -- , get and store the names and the eye color in arrays )
  CR ." Enter your name: "
  ME 14 ENTER                       ( get your name )
  CR ." Enter her name: "
  NAME 14 ENTER                     ( get her name )
  CR ." Enter her eye color: "
  EYES 12 ENTER                     ( get her eye color )
  ;

: LETTER ( -- )
  CR CR CR CR
  ." Dear "
  NAME 14 -TRAILING TYPE
  ." ," CR
  ." I go to heaven whenever I see your deep "
  EYES 12 -TRAILING TYPE
  ."  eyes.  Can " CR
  ." you go to the movies Friday?"
  CR 30 SPACES
  ." Love,"
  CR 30 SPACES
  ME 14 -TRAILING TYPE CR
  ." P.S.  Wear something "
  EYES 12 -TRAILING TYPE
  ."  to show off those eyes!"
  CR CR CR
  ;
```

Type
`HorizontalASCIItable`
or
`VerticalASCIItable`

to display the complete IBM PC character set in two different forms. Many of the graphic characters are interesting because they allow the user to draw fairly nice forms on the screen for business applications. These instructions are also handy if you want to use the graphic characters to construction unique screen displays.

### A Love Letter

Figure Five presents a very interesting example adapted from Leo Brodie's *Starting Forth*. The program allows you to print a love letter inviting your girl friend to see a movie. After entering the code, type `VITALS` and enter all the information requested; then type `LETTER`. You will see the very moving letter.

In this example, we also introduce several important Forth instructions dealing with strings and memory:

`EXPECT` ( addr n -- )
Accept n characters from the keyboard and store them at memory location addr. If the string is terminated by a return, the number of characters accepted is stored in the variable `SPAN`.

### ASCII Character Table

In most computers, characters, numerals, and punctuation are represented in eight-bit bytes. The American Standard Characters for Information Interchange (ASCII) use byte values from 32 to 127 to represent printable symbols. These symbols can be printed on the terminal screen and on any standard printers. Outside of this range, symbols are represented differently by different computers. The IBM PC displays graphic symbols when byte values outside the above range are sent to the screen. The code in Figure Four allows you to display the regular character set, with the graphic characters, in a nicely formatted table.

`ALLOT` ( n -- )
Allocate n bytes of memory following a variable, creating an array to store a string of characters.

`FILL` ( addr n char -- )
Fill a memory array, n bytes long from memory addr, with the character char.

`-TRAILING` ( a n1 -- a n2 )
Modifiy string length n1 to exclude trailing spaces.

Notice that the string arrays `ME`, `NAME`, and `EYES` must be cleared to spaces before calling `EXPECT` in order to enter new information, because we don't know how many characters will be entered into these arrays. After the information is entered, the strings can be retrieved using `-TRAILING` to strip off the trailing spaces.

### Number Input

On many occasions, it is necesary to ask the user to type a number as a string. After the string is accepted, we have to convert it into a number and push it on the data stack for subsequent instructions to use. The primitive instruction to convert a numeric string to a binary number is CONVERT—see Figure Six-a.

With this useful instruction, we can write the game "Guess a Number" shown in Figure Six-b. Type GUESS to initialize the game, and the computer will entertain a user for a while. Note the use of the loop structure:

```
BEGIN
<repeat-clause>
( f ) UNTIL
```

You can jump out of the loop by typing the instruction EXIT, which skips all the instructions in a Forth definition up to ; (the semi-colon)—thus terminating execution of the current definition and continuing to the next definition.

**Figure Six-a.** How CONVERT works.

```
CONVERT ( d1 addr1 -- d2 addr2)
Convert the string at addr1 to a double integer and add it to d1.
Leave the sum as d2 on the stack; addr2 points to the first non-digit
in the string.
```

An example of how to ask the user for a number:

```
: GetNumber ( -- n )
  CR ." Enter a Number: "          ( show message )
  PAD 20 ENTER                     ( get a string )
  0 0 PAD 1 - CONVERT              ( convert string to a double )
  2DROP                            ( leave a single integer on stack )
  ;
```

**Figure Six-b.** Number-guessing game.

```
: InitialNumber ( -- n , set up a number for the player to guess )
  CR CR CR ." What limit do you want?"
  GetNumber                        ( ask the user to enter a number )
  CR ." I have a number between 0 and " DUP .
  CR ." Now you try to guess what it is."
  CR
  CHOOSE                           ( choose a random number )
  ;                                ( between 0 and limit )

: Check ( n1 -- , allow player to guess, exit when the guess is correct )
  BEGIN   CR ." Please enter your guess."
          GetNumber
          2DUP =                   ( equal? )
          IF      2DROP            ( discard both numbers )
                  CR ." Correct!!!"
                  EXIT
          THEN
          OVER >
          IF      CR ." Too High!"
          ELSE    CR ." Too low."
          THEN    CR
  0 UNTIL                          ( always repeat )
  ;

: Greet ( -- )
  CR CR CR ." GUESS A NUMBER"
  CR ." This is a number guessing game.  I'll think"
  CR ." of a number between 0 and any limit you want."
  CR ." (It should be smaller than 32000.)"
  CR ." Then you have to guess what it is."
  ;

: Guess ( -- , the game )
  Greet
  BEGIN   InitialNumber                    ( set initial number)
          Check                            ( let player guess )
          CR CR ." Do you want to play again? (Y/N) "
          KEY                              ( get one key )
          32 OR 110 =                      ( exit if it is N or n )
  UNTIL
  CR CR ." Thank you.  Have a good day."   ( sign off )
  CR
  ;
```

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*.

# The Point of No Return

Mike Elola
San Jose, California

Most professional programmers have abandoned interpreted languages in favor of languages with highly evolved compiling systems. Forth's compiling tools remain simple by comparison.

The ability to merge independently compiled code has been a widely appreciated strategy employed by these compiled languages. The term used to describe the strategy is *separate compilation*. Function libraries are a trivial extension of that particular strategy for developing applications.

Interpreters usually do not compare favorably to compilers, because they perform a translation step at each run of the program. Besides the ensuing performance problems, there are other perceived problems for interpreters. For example, source code is considered a troublesome format in which to distribute applications.

The key virtue of an interpreted language is the hardware independence it accords to source code. Independence from operating systems, processing units, and particular peripheral devices are possible benefits. A "standard" interpreter that runs on microprocessors inside a wealth of devices creates a standard way of controlling those devices. It also accords the source code a considerable level of transportability between the devices. The most favorable conditions for transportability occur when one company controls the language definition. Adobe is the company in control of PostScript, an interpreted page-description language.

A PostScript description of a printed page always causes the same page to be printed. The page is printed faithfully, regardless of the computer platform sending the job, and regardless of the type of PostScript device receiving the job—although quality and print resolution may vary. The PostScript language improves the transportability of printers, helps simplify device drivers and other applications on the host computer, and helps make families of devices more compatible, including printers, typesetters, and plotters.

To avoid re-translation of source code each time a program is run, compilers capture the translated source code into a new file. The new file is the executable file. Besides running faster, such an executable file runs with less supporting software. (The processing hardware suffices to interpret the low-level instructions generated by most compilers.)

Major benefits are associated with compiled applications. One of them is speed. Another benefit is that the form that the executable file takes is not source code. While a different executable file is required for each different hardware instruction set, this has not proved to be a liability. There are many compiler vendors helping to ensure that you can write a program in the compiled language of your choice and still produce executable files to suit your choice of computer hardware.

### Divide and Conquer

Separate compilation is the feature of modern compiling systems that supports the partitioning of programs during development. It allows each subdivision of a program to be independently refined.

Each translated subdivision is stored in a form that is neither a source format nor an executable format. Separate compilation introduces a new file format that is suitable for processing by a link editor, or "linker." The linker can process the translated subdivisions of a program to create an executable file. I am calling the format of the intermediate files produced by such compilers a "linker-ready" file format. See Figure One for an illustration of the processing involved.

Each linker-ready file contains routines that can be thought of as services, particularly if the corresponding subdivision of the program is stable. The clients for those services are any other subdivisions of the program that need to use the services. It's possible for the service and client roles of various subdivisions to alternate over time, but a thoughtful design should eliminate the need to do so.

Once translated to a linker-ready format, the stable subdivisions of the program need only be processed by the linker. The remaining source code is changed and recompiled until it is processed without error. (The interfaces between the service and client routines must not be violated as changes are made.) Then the linker can create a new version of the whole program for testing.

The advantage of a type-checking compiler is that it can certify that the interfaces to precompiled routines are strictly followed. Although the service routines are not repeatedly

recompiled, a type-checking compiler normally processes a source language description of the interface to each precompiled function.

The header file should contain the names of each of the precompiled routines, any parameters they use, and the data types of those parameters. A header file usually resides close to the source code file that actually defines the routines. Sometimes you may have no other source code but the header file, such as when you use a library for which source code is not included.

The compiler's processing of these interface descriptions does not generate code for the linker-ready file, nor for any other file. The interface description files exist merely so the compiler can detect misuses of the interfaces to the precompiled functions inside existing linker-ready files.

(To change a function's interface, several source language files must be brought into agreement: the source file for the service function, the source file for the interface description, and all the source files containing client functions. Then any linker-ready files that were based upon the old interface must be deleted and regenerated. Finally, the executable file can be rebuilt.)

Because of this layered approach to the creation of an executable file, the linker never needs to process the syntax or grammar of any high-level language (nor any assembly mnemonics). For each processor family, associated compilers transform source code in various languages into roughly equivalent types of linker-ready files.

Because it works with previously translated code, the linker lends support for development using a mix of high- and low-level programming languages. Accordingly, assemblers often rely on subsequent linker processing to generate executable files.

Some of the problems you confront for mixed language development are also problems for same-language development through the auspices of separate compilation. For example, the size of integers passed between service functions and their clients must be the same. Because PC-based applications can be compiled to a tiny, medium, or large memory model, addresses passed between service functions and their clients must be based upon the same memory model.

Other troubles for mixed-language development include (1) parameter-passing conventions used by the service functions created in one language as opposed to the conventions used by the client functions created in a different language; and (2) the library file format used by one language compiler as opposed to that used by a different compiler.

Some mixed-language pitfalls are not insurmountable. Certain problems may be overcome by supplying "glue" around the client code that calls the service. The glue must resolve any differences in the run-time assumptions made by the service functions and their clients.

### A Rainbow of Storage File Needs

To better grasp how compiling technology has evolved, let's reexamine the files required by an interpreter, by a basic compiler, and by a modern compiling system. Generally, the process of creating executable files has become more piecemeal, and the ease-of-use of basic compilers and interpreters has been abandoned.

The interpretation strategy uses just one file containing code that serves all the required purposes. The source code for an application and the executable file are the same file.

Progressing to a compilation strategy, two files must be maintained: (1) a source code file; and (2) an executable file.

Progressing from all-at-once compilation to separate compilation, three types of files must be maintained: (1) several source code files; (2) several linker-ready files produced by compilers or assemblers; and (3) an executable file produced by a linker.

(To anticipate future trends, consider this scenario: To progress from separate compilation with all-at-once linking to separate compilation and dynamic linking, no additional files are necessary at compile time. However, the main executable file may later require the linking of it and accompanying linker-ready files. Therefore, the end-user computing platform must be equipped with suitable "linking" software and linker-ready files to successfully run programs that take advantage of this feature.)

The maintenance of many files is the price paid for the improved ability to partition programs. Nevertheless, such partitioning helps support greater code reuse as well as parallel development by several programmers. The efficiency gained from these practices is often a critical factor contributing to the success of a project. Very small programming projects may be the only cases where separate compilation hinders more than it helps.

### Taking Up Collections

The modern compiler consists of two or more layers of tools, as well as collections of files maintained by you or by various build processes (see Figure One).

The raw material for building an application is a pool of files that you bring together. One or more files in the pool may already be in a linker-ready format. Perhaps one file was reused from another project. Perhaps no changes will be required for that item. If so, access to the associated source code is not a requirement.

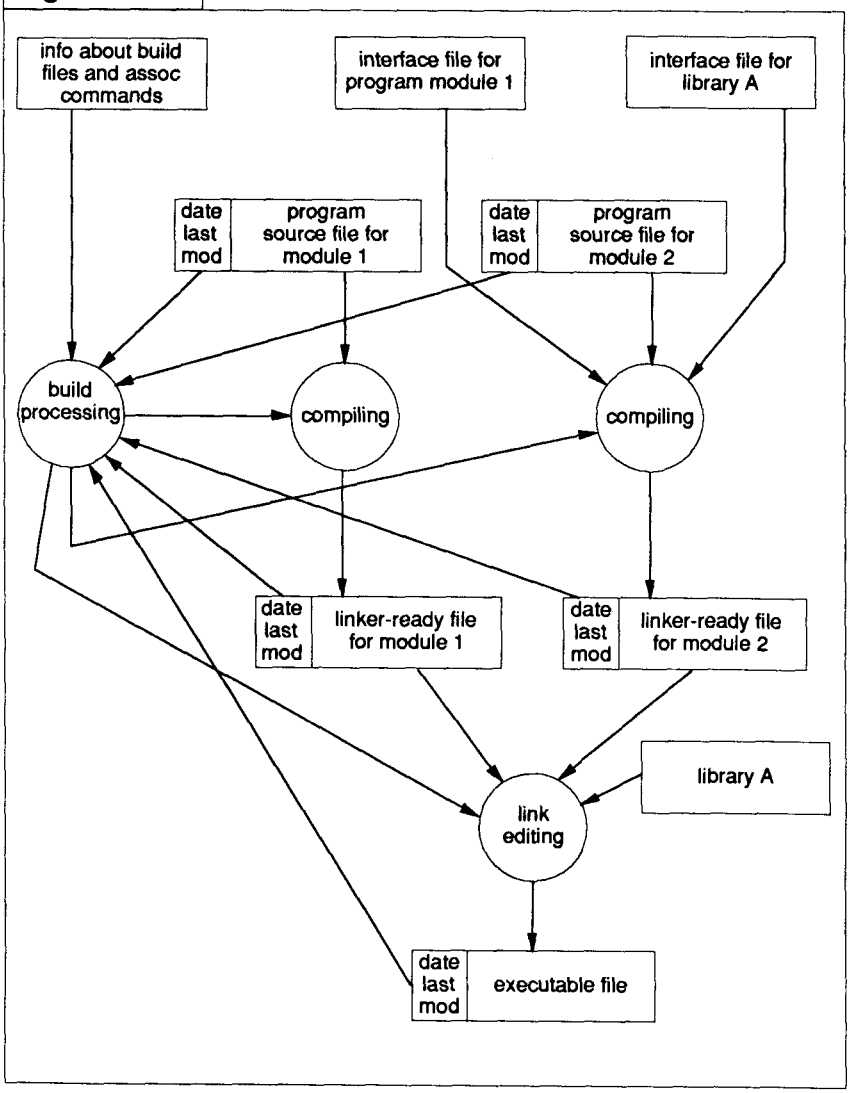
The pool of files must be processed in an appropriate manner so that they can generate a pool of linker-ready files. The processing is carried out by compilers or assemblers, and possibly by various preprocessors or code generation tools. Once the pool of files has been processed correctly, a corresponding pool of linker-ready files is created that the linker converts into an executable file.

A build processing utility such as "make" uses a description of the files and the processes for building a particular executable file, as shown in Figure One. By reading the last-modified date stamped on each file, as well as using the information in the make source file, the make utility can determine the minimum processing required to create a new executable that is up-to-date relative to its subcomponents.

### Well-Traveled Paths to Code Reuse

Through the strategy of separate compilation, several different applications may be produced by linking different combinations of linker-ready files. Some subdivisions of an

**Figure One.**




application may be used repeatedly in a variety of applications. A more common term for the subdivisions of applications is "modules." When referring to the linker-ready file for a module, the term "object module" is frequently used.

Linker-ready files that are reused can be viewed as libraries because their handling is more alike than different. As shown in Figure One, the compilng of module 2 requires two interface files as well as the source code for module 2. One of the interface files serves the library, and one serves module 1. Once module 1 is stable, the make utility uses the same minimal processing for it that is used for library A. The minimum processing for module 1 eliminates the compilation step because the previously created object module remains up-to-date.

The modules that are reused the most are usually collections of routines that perform one type of service, such as string processing. To make these collections of routines easier to reuse—either individually or as a group—additional development tools can be used to repackage them as library files.

Libraries involve a slight elaboration of the feature of separate compilation. First, they establish an alternative format for files that are considered linker-ready files. Second, they are processed differently by the linker.

Routines that reside in normal linker-ready files are included in the executable file by the linker. Routines that reside in library archive files are included in the executable only if the linker determines the need for them by scanning the other linker-ready files that comprise the application. If a routine in the library archive is referenced by the external code in a normal linker-ready file, the linker extracts its precompiled code from the archive and includes it in the executable file. If a routine in the library is not referenced by external code, it is not included in the executable file (unless an already extracted routine depends on it).

At least for library files, the linker must determine which routines are called by other routines, accounting for all the interdependencies among the routines. To speed this process, traversal information included inside the archive usually encodes all the interdependencies that exist among the library routines.

Most of the content of the linker-ready files is dictated by hardware rather than high-level languages. This helps make libraries fundamentally language-independent. The "intermediate code format" is much closer to the machine language than to any high-level language. Even if we are considering the linker-ready file for the Standard C Library, we cannot assume that its language of origin is C. (Was that a challenge you heard?)

The source language in which accompanying interface descriptions are supplied tends to wed a library to a high-level language. However, a vendor is able to add support for another programming language with a minor addition. By supplying several interface (header) files—one for each different language—one object module is allowed to serve multiple programming languages. So a Pascal compiler, if fed a suitable "interface" declaration file, could perform type checks against the parameters being passed to a C function for which no defining source code was submitted.

### Your Turn Now

The sophistication of library tools, and of the underlying technique of separate compilation, deserves our attention and our assimilation into Forth. For good reasons, there is never likely to be a retreat to simpler methods or tools. So we should plan on adding this support to Forth as a complement to Forth's native compiling provisions.

We have explored the advances that have occurred with regard to application development tools, particularly in terms of compiling strategies. I hope I have not made any major guffaws in this attempt to create a tutorial out of my limited understanding of this material. If so, I apologize. (Okay, so I might have stretched the truth when I said that three kinds of files were needed for support of separate compilation. I overlooked the header files.)

## FIG Chapters

*The Forth Interest Group Chapters listed below are currently registered as active with regular meetings. If your chapter listing is missing or incorrect, please contact the FIG office's Chapter Desk. This listing will be updated regularly in* Forth Dimensions. *If you would like to begin a FIG Chapter in your area, write for a "Chapter Kit and Application."*

*Forth Interest Group*
*P.O. Box 2154*
*Oakland, California 94621*

### U.S.A.

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

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

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

- **CALIFORNIA**
  **Los Angeles Chapter**
  4th Sat., 10 a.m.
  (except Nov., Dec.)
  Joslyn Center
  339 Sheldon, El Segundo
  John LuValle
  (818) 797-1820
  Howard Rogers
  (301) 532-3342

  **North Bay Chapter**
  2nd Sat.
  12 noon tutorial, 1 p.m. Forth
  2055 Center St., Berkeley
  Leonard Morgenstern
  (415) 376-5241

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

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

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

  **Silicon Valley Chapter**
  4th Sat., 10 a.m.
  Applied Bio Systems
  Foster City
  John Hall
  (415) 535-1294

  **Stockton Chapter**
  Doug Dillon (209) 931-2448

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

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

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

- **ILLINOIS**
  **Cache Forth Chapter**
  Oak Park
  Clyde W. Phillips, Jr.
  (708) 713-5365

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

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

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

  **Fairfield FIG Chapter**
  4th Day, 8:15 p.m.
  Gurdy Leete (515) 472-7782

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

- **MASSACHUSETTS**
  **Boston FIG**
  3rd Wed., 7 p.m.
  Bull HN
  300 Concord Rd., Billerica
  Gary Chanson (617) 527-7206

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

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

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

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

- **NEW JERSEY**
  **New Jersey Chapter**
  Rutgers Univ., Piscataway
  Nicholas G. Lordi
  (908) 932-2662

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

- **NEW YORK**
  **Long Island Chapter**
  3rd Thurs., 7:30 p.m.
  Brookhaven National Lab
  AGS dept.,
  bldg. 911, lab rm. A-202
  Irving Montanez
  (516) 282-2540

  **Rochester Chapter**
  Monroe Comm. College
  Bldg. 7, Rm. 102
  Frank Lanzafame
  (716) 482-3398

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

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

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

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

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

  **Dallas Chapter**
  4th Thurs., 7:30 p.m.
  Texas Instruments
  13500 N. Central Expwy.
  Semiconductor Cafeteria
  Conference Room A
  Warren Bean (214) 480-3115

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

- **VERMONT**

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

- **VIRGINIA**

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

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

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

- **WISCONSIN**

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

## INTERNATIONAL

- **AUSTRALIA**

**Melbourne Chapter**
1st Fri., 8 p.m.
Lance Collins
65 Martin Road
Glen Iris, Victoria 3146
03/889-2600
BBS: 61 3 809 1787

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

- **BELGIUM**

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

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

- **CANADA**

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

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

**Southern Ontario Chapter**
Quarterly: 1st Sat. of Mar., June, and Dec. 2nd Sat. of Sept.
Genl. Sci. Bldg., RM 212
McMaster University
Dr. N. Solntseff
(416) 525-9140 x3443

- **ENGLAND**

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

- **FINLAND**

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

- **GERMANY**

**Germany FIG Chapter**
Heinz Schnitter
Forth-Gesellschaft e.V.
Postfach 1110
D-8044 Unterschleißheim
(49) (89) 317 37 84
Munich Forth Box:
(49) (89) 871 45 48
8N1 300, 1200, 2400 baud
e-mail uucp:
secretary@forthev.UUCP
Internet:
secretary@Admin.FORTH-eV.de

- **HOLLAND**

**Holland Chapter**
Maurits Wijzenbeek
Nieuwendammerdijk 254
1025 LX Amsterdam
The Netherlands
++(20) 636 2343

- **ITALY**

**FIG Italia**
Marco Tausel
Via Gerolamo Forni 48
20161 Milano

- **JAPAN**

**Japan Chapter**
Toshio Inoue
University of Tokyo
Dept. of Mineral Development
Faculty of Engineering
7-3-1 Hongo, Bunkyo-ku
Tokyo 113, Japan
(81)3-3812-2111 ext. 7073

- **REPUBLIC OF CHINA**

**R.O.C. Chapter**
Ching-Tang Tseng
P.O. Box 28
Longtan, Taoyuan, Taiwan
(03) 4798925

- **SWEDEN**

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

- **SWITZERLAND**

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

## *Why FIG Chapters?*

*See the editorial on this issue for news from the new chapter in southern Wisconsin...*

## SPECIAL GROUPS

- **Forth Engines Users Group**

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

("Back Burner," continued from page 39.)

nents of the Forth environment reside in application RAM, so they must be created anew each time the application hardware is powered up or reset.

Once initialization is complete, the Forth virtual machine must be started (i.e., activated). The machine is started by assigning it a word—the application program—to execute. Once started, the machine never stops, apart from reset, a system crash, or power failure. The programmer must ensure that the machine *always* has something to do. He does this by writing the application program in the form of an endless loop.

### The Bare Essentials

We shall assume the following resources:

- the application hardware: a ROM-based computer, such as the 8051 trainer presented in column #5, with serial interface
- the development system: a computer of the IBM PC genre, providing mass storage (fixed disk) and serial interface, as well as keyboard and text-oriented screen for operator interface
- (alternative A) a cross-assembler and a compatible text editor, both running on the development system; the cross-assembler produces code which is to execute on the application hardware
- (alternative B) a Forth environment, *sans metacompiler,* running on the development system

Although we will use as our application platform the 8051 trainer, the term *application* does not imply a limitation of capability: in general, the application system may itself be a development system, and the Forth environment implemented on the application hardware may include a metacompiler. Indeed, the application system and the development system may be identical, even to the extent that they are one and the same. This latter case occurs particularly if you are implementing Forth without having access to a Forth development environment, or if you are reconfiguring an existing Forth environment.

### The Scenic Route

Given these resources, there are two alternative routes by which we may reach our goal.

What at first *appears* to be the most straightforward approach utilizes assembly language to define each component of Forth for the application environment. The assembler or cross-assembler runs on the development system, producing code which is to run on the application hardware. Once the assembly source has been debugged and a Forth environment has successfully been implemented on one application platform, the same source code may be used as the basis for implementing Forth on other platforms.

The assembly language approach is used by R.G. Loeliger in his book *Threaded Interpretive Languages* (Byte Books, Peterborough, New Hampshire, 1981). The approach is also the method originally used to distribute fig-Forth circa 1980 and is, in principle, the approach used in the eFORTH system of C.H. Ting. If you take the trouble to inspect Loeliger's book or one of the fig-Forth listings and the accompanying implementation notes, it should quickly become apparent that the assembly language approach is rather convoluted, and not for the faint of heart.

Interestingly, Ting utilizes the Microsoft 80x86 macro assembler, rather than a cross-assembler, regardless of the application hardware. If, however, a cross-assembler is used, porting (transporting) Forth between various platforms will be considerably simplified when the cross-assembler is of the "universal" (i.e., table-driven) variety. Such assemblers retain the same syntax, irrespective of the processor; the table for each processor provides a set of processor-specific opcodes. The process of converting an assembly language file for one assembler to work with another assembler of incompatible syntax can involve long hours of tedious editing. Representative universal assemblers are TASM (shareware by Speech Technology, Issaquah, Washington) and Cross-32 (Universal Cross Assemblers, Nova Scotia, Canada).

The metacompilation approach makes use of Forth in the development environment to define Forth for the application environment. The approach is not generally understood and has been shrouded with an undeserved mystique. Many consider it an esoteric technique which is strictly within the domain of the Forth guru. This is a false impression. Metacompilation is the most simple, most direct method of porting Forth from one system or platform to another. Granted, you must know what you are doing. However, once you understand what is going on, the process frees you from a confusing mass of detail and from a multitude of constraints.

### Follow the Yellow Brick Road

With metacompilation, the procedure is as follows: within the Forth development environment, we write:

- an assembler for the instruction set of the application processor
- a metacompiler which incorporates the assembler
- application source (a mixture of Forth code words and high-level Forth words)
- application startup code, i.e., source for initialization of the application system (Forth code words)

While we are in the Forth development environment, we load the metacompiler, just as we would load any other Forth application which runs on the development system. We then load the source code for the application. The metacompiler operates on the source code as it is loaded, producing application *object code.* Finally, we copy the application object code from the development system to ROM or floppy disk.

Although the assembly language approach is adequate for the task of creating a new Forth environment, metacompilation facilitates the process by placing at the disposal of the programmer the full resources of the Forth language. Specific advantages of metacompilation over the assembly language approach include:

- The word compilation mechanism inherent in Forth is utilized to create heads and parameter fields within the

application dictionary.

- The programmer is free to implement whatever assembler syntax he finds convenient.
- Diagnostic words may be defined and executed interactively.

### Checking Out the Competition

The book *Embedded Controller Forth for the 8051 Family* by W.H. Payne (Academic Press, Boston, 1990) follows the metacompilation approach. I hesitate to recommend the text, inasmuch as I find it rather disorganized and unapproachable. Moreover, I disagree with a number of Payne's statements and the conclusions they imply. One paragraph in particular illustrates my objections; in it, Payne says:

> The text of this book is its least important aspect. The code presented in this book warrants your greatest study. Charles Moore's genius is brought to you by members of the Forth Interest Group. Jerry Boutelle's genius with metacompilers allows us to generate Forth from source. You can figure out what Charles Moore had in mind fairly easily. Understanding what Jerry is doing in the metacompiler baffles the best programmers.

In this *On the Back Burner* series on metacompilation, the code is the *least* important aspect. Code fragments which I present are not optimized (nor are they guaranteed to work correctly). They are only examples, to facilitate your comprehension. If you understand the text, you will be able to write your own code, and your code will, in most cases, be superior to that of the examples.

It should be noted that, whereas the metacompiler to which Payne refers apparently is straight from the hand of Boutelle, the fig-Forth listings to which Payne refers do not necessarily convey the genius of Chuck Moore, inasmuch as they are removed by several generations from his direct influence.

A metacompiler is not *inherently* a complex application, nor is metacompilation *inherently* a complex process. A properly designed metacompiler should not baffle a programmer. A design is a solution to a problem. A fundamental criterion of excellence in design is simplicity. Simplicity in design is the hallmark of a designer who understands the problem.

Designs of the Rube Goldberg variety typically indicate either that the designer did not understand the problem at hand, or that he faced overwhelming constraints. (It may well be that adding metacompilation capability to dudFORTH admits of no good solution.)

The foregoing is not meant to imply that every code segment should be *transparent*. As with a mating combination in chess, excellent code may sometimes be abstruse, although it often is the essence of simplicity. Code of this category should *always* be documented.

Admittedly, my effort in this metacompilation series is parallel to that of Payne. The fact that I chose the same application platform (the 8051) as did Payne is purely coincidental, a matter of circumstance. However, my purpose is not to duplicate Payne's treatment of the subject. I believe that Payne's approach is unnecessarily complex: there is too much source code and the procedures are too involved. After all, the complete body of source code for native-mode polyFORTH for the DEC PDP/LSI-11—a microcomputer with the instruction set of a minicomputer (a really marvelous machine, a joy to program)—occupies only 83 sparsely populated 1024-byte screens. Moreover, the polyFORTH system has an integral metacompiler.

### Pulling Yourself Up by Your Own Bootstraps

If you find yourself without access to a Forth development environment, it would not be unreasonable to resort to the assembly language approach in order to *build* a Forth development environment, so as to enable further development to be conducted via metacompilation. In fact, Payne's book assumes just such a scenario, and Payne provides code for virtually everything in the way of software needed by those taking this tack.

### Whence?

Since this column (#9) is devoted to the matter of orientation, perhaps we should recall what we have covered in previous columns. Column #3 announced a focus upon the field of embedded systems, with special emphasis on the subject of metacompilation. Column #4 described and provided listings for an 8051-family assembler, a key component of a metacompiler. The fifth column (no pun intended) presented plans for an 8051-family single-board computer for use as a metacompilation trainer. Column #6 consisted of a foray into the terminology associated with metacompilation. Column #7 dealt with the Forth vocabulary structure and with the utilization of vocabularies in the process of metacompilation. Column #8 discussed pointers and some of the intricacies of stack implementation.

### Whither?

What's in store? If we simply sit down in front of a terminal and begin typing

```
: METACOMPILER
```

we won't get very far unless we have in our minds a clear picture of the elements which comprise a Forth system and how each element fits into the whole. Thus, you see the reason for digressions to discuss matters such as assemblers and vocabularies. Our next investigation will explore threading and the inner interpreter. Stay tuned. Meanwhile, drop me a line.

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 welcomes clients who offer him the opportunity to work with Forth. He can be reached by phone at 713-461-1618, by facsimile at 713-461-0081, by mail at 8609 Cedardale Drive, Houston, Texas 77055, or on GEnie (address RUSSELL.H).

*(Character classification, continued from page 8.)*

the source operand is somewhere in the lookup table at `_ctype`, as indicated by that register.

Had we wanted to emulate the C code exactly, the place to implement the necessary `1+` would have been to change the byte displacement value to a 1. This would have added two bytes to the size of the table: the byte at `_ctype`, and one byte of wastage due to word alignment required by the processor. But, unlike the C macros, it would not have added any instructions to the code.

Line ten places the results on the stack, and line 11 sets our return to the calling code.

Note that the result is not necessarily a one, minus one, or a zero. The result is the product of the and between the bit mask in the classifier word and the mask in the array. This is not a "standard" implementation. However, the user may find the resulting bit mask useful for further processing, and the zero or non-zero result is useful.

Even with fastForth's efficient compiler, the assembly language version of the code is somewhat faster than the high-level version, and considerably smaller.

Screen 1926 uses the defining word `CLASSIFIER` to build several classification words, analogous to the macros in the second group in ctype.h. There are single-bit mask words, such as `ISLOWER` and `ISUPPER`. Masks may be or'd together to produce larger classifications. An example of the latter is `ISALPHA`, which tests for a character being either an upper- or lower-case character.

### Results

The exercise produced clean code, easily understood (with the possible exception of the assembly language), and very fast. The code probably takes more space than the code it was written to replace, but the user may find that the added flexibility is well worth it.

The whole process—from being dissatisfied with Mr. Schaaf's code, reverse engineering the C code, writing the code in Forth (including the assembly language), and testing the Forth code—took less than two hours. It took longer to write this article.

### Availability

This code is released to the public domain. Enjoy it in good health, and toast the health of contributors to the public domain from time to time.

---

Charles Curley, a long-time Forth nuclear guru, has his own private postal code: 82717-2071.





## ADVERTISERS INDEX

## On the Back Burner #9

# Who's on First?

*Conducted by Russell L. Harris*
*Houston, Texas*

Column #7 generated a brief flurry of response, a total of ten contacts (from a circulation of 1600?). I need feedback on a continual basis, in order to match the presentation to *your* background and level of understanding. Particularly, I need to hear from anyone who finds details of a presentation unclear. If you don't quite understand what's going on, you're probably not alone. However, since we're not in a classroom where I can look for blank faces and puzzled expressions, *you* must take the initiative and contact me whenever you hit a snag. I will try to get you over the immediate hurdle; then, in a subsequent column, I will backtrack and elaborate on those areas you found unclear.

### Breaking the Ice

Don't let things pile up: In *any* classroom environment, as soon as you see that you're lost, interrupt the lecture and state the area in which you are confused. You will be surprised how many of your fellow students are confused on the same point, but were too embarrassed to be the first to admit it. By speaking out, you benefit everyone: you, your classmates, and the teacher. Understanding is built upon understanding: Fundamental concepts which are not mastered at first encounter can quickly snowball into insurmountable barriers.

An aside: A lecturer who disallows interruptions, insists upon unquestioning submission to his "authority," or brands as heresy any challenge of "facts" or theories he presents, should immediately be suspect. Truth welcomes—nay, *demands*—scrutiny. In this regard, you may find interesting Albert Einstein's essay "On Education" (1936).

Regarding the level of presentation, I intend this column to be easily comprehensible by anyone who has successively navigated the text *Starting Forth* by Leo Brodie/FORTH, Inc. Our current topic, metacompilation, is not a terribly advanced subject!

Another aside: I doubt if even half those who *think* they have mastered *Starting Forth* have in reality done so. Although the topics covered in *Starting Forth* are not difficult, the book itself is deceptively elementary. Unless you have personally worked through each of the exercises, you probably have failed to comprehend a number of critical concepts.

### A Panorama

Several readers have expressed confusion concerning both our ultimate goal and the means of achieving it.

Our goal is to implement a Forth environment on a computer we term the *application hardware*. We accomplish our goal by using a computer, the *development system*, to generate code which, when executed on the application hardware, first *creates* a Forth environment and then *activates* the environment.

Having generated executable application code in the development environment, we copy the code to disk or to ROM, according to the configuration of the application hardware. On hardware such as the IBM PC, application code typically executes from RAM and must be loaded into RAM from disk. On hardware such as the 8051 single-board computer (SBC) presented in column #5, application code executes directly from ROM.

### The Kick-Off

The segment of the application code which first receives control of the processor is termed the *initialization code*. In our usage, the term will embrace creation and activation of the Forth environment, as well as configuration of the hardware.

Most computer systems contain one or more hardware components which can assume a variety of configurations (i.e., behaviours). The processor, the serial interface adapter, and the parallel interface adapter typically are of this nature. Serial communication parameters and the direction (i.e., input or output) of parallel ports are representative of behaviours which vary with component configuration. Originally, component configuration was accomplished by jumpering pins of the respective integrated circuit. Now, however, configuration is typically done under program control, by loading specified values to registers within the devices.

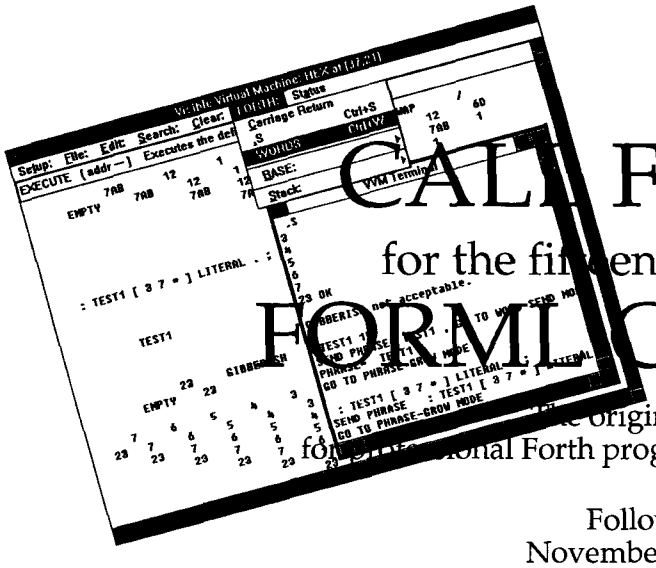
Hardware configuration is the first task performed by the initialization code. After configuring the hardware, many initialization routines perform a checksum or cyclic redundancy check (CRC) on system ROM and a read/write diagnostic on system RAM.

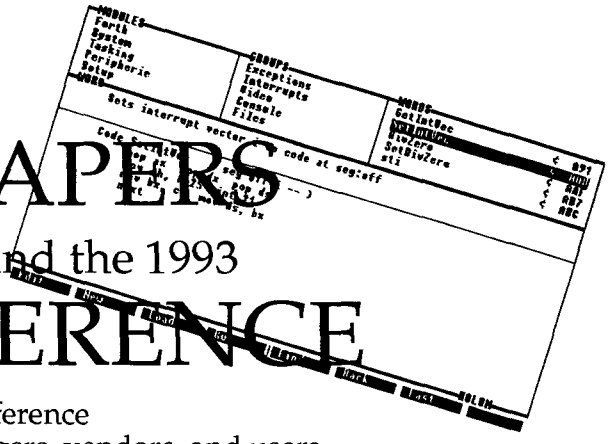
If the application hardware is a disk-based system such as the IBM PC, the ROM BIOS furnished with the PC is responsible for hardware initialization. Thus, the initialization portion of Forth application code running on a PC need concern itself only with setting up and activating the Forth environment. If the application hardware is a ROM-based system such as our 8051 trainer, the application code we generate will both configure the hardware *and* set up and activate the Forth environment.

### The Set-Up

The process of setting up the Forth environment involves reassignment of interrupt vectors, definition and initialization of pointers which bring into existence the stacks, buffers, dictionary, etc., and definition and initialization of pointers which constitute the user area of OPERATOR. These compo-

*(Continues on page 34.)*





# CALL FOR PAPERS

for the fifteenth annual and the 1993

# FORML CONFERENCE

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

Following Thanksgiving
November 26-November 28, 1993
Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California U.S.A.

## Theme: **Forth Development Environment**

Papers are invited that address relevant issues in the establishment and use of a Forth development environment. Some of the areas and issues that will be looked at consist of networked platform independence, machine independence, kernel independence, development system/application system independence, human-machine interface, source management and version control, help facilities, editor development interface, source and object libraries, source block and ASCII text independence, source browsers including editors, tree displays and source data-base, run-time browsers including debuggers and decompilers, networked development/target systems.

**Mail abstracts of approximately 100 words by September 1, 1993.**
**Completed papers are due November 1, 1993.**

Registration fee for conference attendees includes 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.

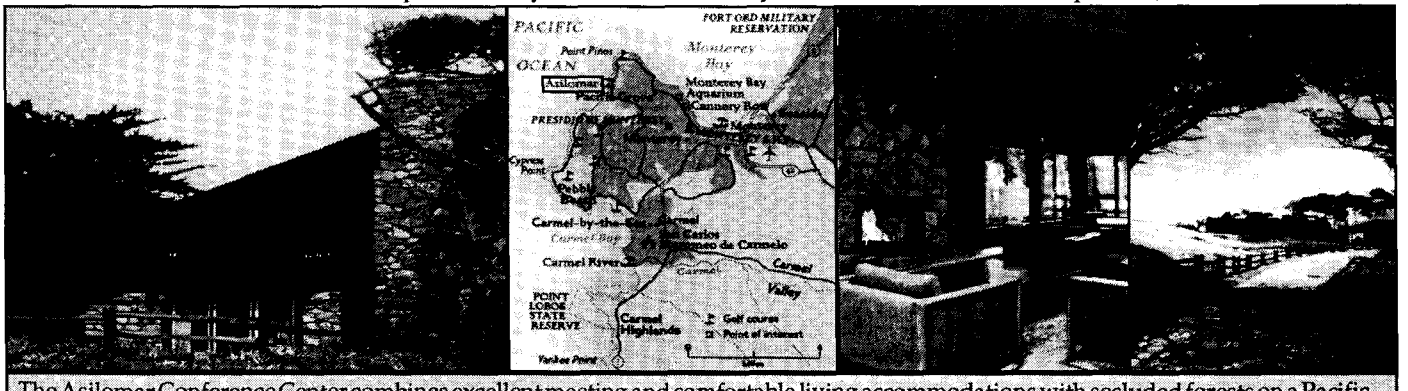
Conference attendee in double room - **$380** • Non-conference guest in same room - **$260** • Children under 18 years old in same room - **$160** • Infants under 2 years old in same room - **free** • Conference attendee in single room - **$490**

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

John Hall, Conference Chairman

Robert Reiling, Conference Director

Register by calling, fax or writing to:
Forth Interest Group, P.O. Box 2154, Oakland, CA 94621, (510) 893-6784, fax (510) 535-1295
This conference is sponsored by FORML, an activity of the Forth Interest Group, Inc. (FIG).




The Asilomar Conference Center combines excellent meeting and comfortable living accommodations with secluded forests on a Pacific ocean beach. Registration includes use of conference facilities, deluxe rooms, all meals, and nightly wine and cheese parties.