$10 Volume XV, Number 5 January 1994 February

FORTH DIMENSIONS

I Needed It: Mini-Math

Comma'd Numeric Output

Rational Numbers, Vulgar Words

The Visible Virtual Machine

A Forth Development Environment for Embedded Real-Time Control

Contents

Features

Departments

Editorial

Of Awards and Incentives

As I explained to FORML attendees, before rushing home to put this issue to bed, the Forth Interest Group in recent years has generously sponsored cash awards for contests in *Forth Dimensions*. This time, we are pleased to present winning papers on the topic of Forth Development Environments.

★ *First Place:* B. Meuris, V. Vande Keere, J. Vandewege, "A Novel Approach to Forth Development Environments for Embedded Real-Time Control"
★ *Second Place:* Richard Astle, "Forth and the Rest of the (DOS) World
★ *Third Place:* Ellis D. Cooper, "The Visible Virtual Machine"

Richard Astle's paper was printed in our preceding issue; the other two winners appear herein. The additional papers we received had considerable merit, and we hope to present some of them in the near future.

Forth Development Environments was also the theme of this year's FORML conference and, interestingly, the awards committee for that event independently selected two of these *FD* papers for recognition there. (Cooper's paper was not presented at FORML.) Although the criteria for FORML awards may differ from those used in selecting papers for publication in *Forth Dimensions*, those submissions were judged to be of great interest and value to both audiences.

Bill Ragsdale, who presented the awards, will provide general conference coverage later, both in these pages and on the Internet. But in the interests of timeliness, here are the awards that were given for papers at FORML:

★ *Best Presentation of a Work in Progress (shows value and promise—we expect more next year):*
Wil Baden, "Optimization, Macros, I/O Re-direction, and Ecumenicity"
★ *Featured Paper on Forth with Benefit to the non-Forth Community:*
Richard Astle, "Forth and the Rest of the (DOS) World
★ *Outstanding Paper on Applications and Utilities (of interest to most people, timely and useful):*
Richard Wagner, "Simple Mouse and Button Words"
★ *"Blue Ribbon" Award for the Premier Paper on Forth Development Environments (the conference theme):*
B. Meuris, V. Vande Keere, J. Vandewege, "A Novel Approach to Forth Development Environments for Embedded Real-Time Control"

Although I have tried before, and no doubt will again, no description adequately conveys the effect of attending FORML. The association with peers, new friends and old, repeated immersions into very technical (but user friendly) sessions, cross-pollination of ideas, and a small-conference facility that takes full advantage of one of California's most spectacular natural settings, create a stimulating, unforgettable, and rewarding experience. (Even I, notoriously bonded to my homestead, get nostalgic as the conference draws to a close, postponing departure as long as practical.) I encourage you to pre-register *as early as possible* to ensure your seat at the November 1994 FORML.

* * *

Something seems to be happening. Whether it is caused by advance ripples from dpANS Forth, by mention of Forth in other publications, or by the normal (but poorly understood) cycles of public interest, is unknown. Quietly, the Forth language has been getting some interesting attention. This is evidenced by secondhand reports at FIG Chapter meetings and occasionally by telephone calls received at the FIG office—calls from employers looking for Forth programmers and engineers. Out of fairness and objectivity, the FIG office tells me it does not recommend particular programmers, but it can circulate information about job openings to the Forth community. The easiest ways to do that are through on-line venues and FIG Chapters. They were able to help place five Forth engineers with Varian recently, for example, by contacting chapter leaders and asking them to announce the opportunity at their next meeting. This is just one example of FIG membership at work, and one more reason to stay in touch on line and at your nearest chapter.

—Marlin Ouverson

Forth Dimensions
Volume XV, Number 5
January 1994 February

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, 4800 Allendale Ave., Oakland, CA 94619. Second-class postage paid at Oakland, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621-0054."

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!

Aborting with the Same Message?

Dear Mr. Ouverson,

I originally used Forth to do some tremendous business applications for a company on the VIC-20 and Commodore 64! Unfortunately, they were highly specialized, and I had no marketing experience, so nothing ever came of this. I ignored Forth for eight years. The reason I'm back is I'm finally working for myself, and no longer have to justify the languages I choose to anyone.

I am now back up to speed on Forth, I still love it for all the same reasons, am even more dissatisfied with every other language I have used. But it still has the same problems as ever, and it seems the "official community" has even less of a chance than ever of getting it.

Based on my exposure to *every other* language and applications development system I have used: Pascal, C, C++, BASIC (ugh), FOXpro, dBase, Paradox, R:Base, Clipper, and Clarion, I expected by now to be able to buy an applications-development-oriented Forth with the following characteristics:

You need to attract the kind of users who value their time, who simply want a good environment from which to solve problems.

---

1. A fully implemented link to the most common data-management program in the DOS world (dBase III file standard).
2. A resolution to the problem of not being able to type anyone's source code in without understanding more than you care to about the differences between implementations.
3. An editor similar to the one shipped in *any of the above packages*. Does anyone know the history of WordStar-compatible DOS editors? This is *really, really* frustrating. So what if I can reconfigure it—why isn't it the way I expect out of the box? What world do you live in?
4. A way of ordering add-on libraries with:
   a. CUA-compliant menus and windows, dialog boxes
   b. graphics
   c. other file drivers (spreadsheet, word processing)
   d. report writers
   e. printer drivers
   f. fax drivers
   g. TrueType font generator/interface/driver for printers and screens
5. The possibility of being able to fax a client a few lines of fixes and allowing them to recompile an application, achieving painless bug fixes or customization. (This is the most controversial suggestion, I suppose. I think vectored execution and speedy recompilation are the most significant tools for improving and customizing a program, and would bring Forth program development way, way, way out in advance of anything the other guys can do. It means including source code, possibly at additional cost. Two possible ways to make this more palatable: either develop a *shroud*, or trust in the inability of the standard Joe to get in and do anything with Forth. Either way, you are no less protected from theft by copying than you are by distributing the program itself.)
6. Real hopes for being able to run *without modification* on the DOS, Macintosh, and Unix platforms.

What I found instead was that you are still talking about writing your own compilers, that embedded systems is your last toehold, that you can't include the ability to compile when distributing applications, that the company I naively considered the main force behind general acceptance of Forth has a sign on its wall saying, "*Embedded* Forth Controllers for *Industry*, Stupid"—a great niche for a company, but hardly an adequate vision for the kind of acceptance I think the rest of us want.

My suggestions:

1. Split embedded systems from general application work. We need separate user's groups. I am so bored with talk at my local user's group that I no longer go in the hopes of an occasional application-oriented discussion. Embedded systems care about size; DOS users really don't. Embedded systems care about speed; '386 users trying to develop for the main market should put this about 45th on the list.
2. *Obey* the damn *standards*. If you insist on having your `DO LOOP` work differently, *name it differently*. I won't mind seeing `DO-LMI`, `CFA-HSF`, `LOOP-79`, and `ASCII-FPC`, but I really get steamed when I can't type in examples from magazines. If you are thinking, "But it's no big deal if you understand..." just drop it and get with the program. Application programmers have zero, zero, zero interest in keeping straight all the dialects. Might as well go with Clipper.
   a. Now, all true vendors must support all major possibilities. Please don't argue that this is a major hassle, it makes your product *usable*.

(Continues on page 36.)

HARVARD SOFTWORKS

NUMBER ONE IN FORTH INNOVATION

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

Just how good is HS/FORTH? Well, it's already good enough to control mile long irrigation arms to water the nations crops, good enough to control orbiting shuttle experiments, good enough to analyze the nation's blood supply and to control the telephone switching systems. It monitors pollution (nuclear and conventional) and simulates growth and decline of populations. It's good enough to simulate and control giant diesel generator engines and super cooled magnet arrays for particle accelerators. In the army and in the navy, at small clinics and large hospitals, even in the National Archives, HS/FORTH helps control equipment and manage data. It's good enough to control leading edge kinetic art, and even run light shows in New York's Metropolitan Museum of Art. Good enough to form the foundation of several of today's most innovative games (educational and just for fun), especially those with animation and mini-movies. If you've been zapping Romulans, governing nations, airports or train stations, or just learning to type - you may have been using HS/FORTH.

Our customers come from all walks of life. Doctors, lawyers and Indian Chiefs, astronomers and physicists, professional programmers and dedicated amateurs, students and retirees, engineers and hobbyists, soldiers and environmentalists, churches and social clubs. HS/FORTH was easy enough for all to learn, powerful enough to provide solutions, compact enough to fit on increasingly crowded disks. Give us a chance to help you too!

You can run HS/FORTH under DOS or **Microsoft Windows** in text and/or graphics windows with various icons and pif files for each. 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. There are few limits to program size since large programs simply grow into additional segments or even out onto disk. The Tools & Toys disk includes a complete mouse interface with menu support in both text and graphics modes. With HS/FORTH, one .EXE file and a collection of text files are all that you ever need. Since HS/FORTH compiles to executable code faster than most languages link, there is no need for wasteful, confusing intermediate file clutter.

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.**

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

Rational Numbers, Vulgar Words

Gordon Charlton
Hayes, Middlesex, U.K.

There are several different ways of representing fractional numbers in a computer. The suite of words given here implements one of the less common ones, a rational representation.

A rational number is one that can be represented as the ratio of two integers, called the numerator and the denominator. If this sounds familiar, it is because this is how we are first taught to deal with fractions, in primary school. Chances are that you have not dealt with them much since, but you can vaguely recall bits and pieces about multiplying across the diagonal and cancelling out common terms and so on. I knew them as vulgar fractions.

The idea of implementing a system that is taught to seven-year-olds appealed to me, having looked at the complexities involved in floating-point representations. It soon turned out that primary level maths was not sufficient for the task, some number theory is also required. In particular, it is necessary to switch from a rational representation to a continued fraction format and back again during rounding, but that will be described later.

Between accuracy and precision, I would choose accuracy every time.

Although the code presented here will work equivalently on either a 16-bit or a 32-bit system, I shall describe it in terms of an eight-bit Forth, to keep the numbers simple. Feel free to multiply any numbers up to suit your Forth.

The Representation

A rational number is here represented by two cells, the first being completely filled by a sign bit and the numerator, which occupies seven bits. The second cell contains the denominator, which is also seven bits wide. The high bit is unused. This decision was made to allow the full range of Forth operators to be used in the implementation rather than just the signed operators, or just the unsigned ones. It also allows the double-length division that is at the heart of the rounding routine to operate only on signed positive numbers, which allows a more efficient implementation.

Some Extra Integer Operators

In fact we require a very rich set of maths operators, so my first task was to beef up my Forth with some additional maths operators, mostly intended for the double division. I am indebted to Prof. Tim Hendtlass and, in turn, to Prof. Nathaniel Grossman for a good chunk of this code. Other words are derived from work I had done previously. In each case, I have selected the most efficient rather than the most comprehensible. As Hendtlass reworked Grossman's code to iron out a few inefficiencies, so I have reworked the double division to iron out a few more. One is forced to wonder how much further the process can be taken.

However, during the writing/testing phase of development, it became obvious that some of my primitives were in error. In particular, `UM/MOD` and `D-`. `UM/MOD` gives up the ghost with some numbers where the high bit is set. I am not terribly upset by this, as I have a 32-bit Forth and it is very unusual to push `UM/MOD` terribly hard, so I had not previously encountered the bug. Secondly, I use a budget Forth, and it is a sad fact that, in this world, one gets what one pays for. On the other hand, I am fuming about `D-`, which consistently gives the negation of the correct answer. That kind of error should not occur. End of story.

So the first screen gives high-level equivalents of these two words. They are included as it is a policy of mine only to publish code that can be run on my own machine. The other is to try and make it reasonably portable. Hopefully, your `UM/MOD` will be up to the demands we are going to place on it. If it is not, the high-level one given here works, but it is slow. This means I will not be discussing execution speed in any great detail.

I have attempted to name the maths operators according to what I infer to be the convention adopted by the ANSI committee, as I find it satisfactory. `+D/` and `+D/MOD` work only on positive signed integers, which explains the plus sign.

Median Rounding

Plunging on into the code, we find that the first thing required is a rounding routine. There is a problem with a rational representation—the number of digits required to exactly represent a number can grow beyond the bounds of the cell containing it quite frequently, even when kept in a

normalised format. By this, we mean not only that the sign bit be kept in the right place, but that the numerator and denominator be relatively prime and, hence, in their simplest state. Normalising rationals in that respect does slow the process, so we will make it a convention that numbers are held in their normalised form at all times, thereby spreading the delays out evenly through the code. GCD uses the Euclidean algorithm to determine the highest common divisor of the numerator and denominator. In common with many words in the suite, it assumes that only the second on stack is a signed number.

This is used by NORMAL, which converts a vulgar fraction to its simplest form (i.e., 1/2 rather than 26/52). The next key definition, REDUCE, takes rather more explanation.

It is first necessary to realise that a vulgar fraction can overflow in three different ways: by becoming too large, so the numerator overflows; by becoming too small, so the denominator overflows; or by the fraction becoming too complex to be represented within a single representation, whilst still lying within the range of representable numbers. To cater for this, primitives which can increase the size of the numerator or denominator produce intermediate double-length results which are then rounded when the representation becomes too large. REDUCE does this.

If no overflow has occurred, or a large double number can be normalised to single length, then it simply makes it so. Numbers which exhibit the third form of overflow are reduced to that single-length vulgar which most closely approximates the value of the double vulgar passed to it. If

```
\ high level D- UM/MOD ( core versions are incorrect)
: 4DUP ( n n n n--n n n n n n n n) 2over 2over ;
: D- ( d d--d) dnegate d+ ;
: UD< ( ud ud--f)  rot 2dup = not
                   IF  swap 2swap  THEN  2drop u< ;
: (UM/MOD)  ( ud ud--ud u)
            2dup >r >r dup 0<
            IF 2drop 0 ELSE 2dup d+ RECURSE 2* THEN
            -rot r> r> 4dup ud<
            IF 2drop rot ELSE d- rot 1+ THEN ;
: UM/MOD ( ud u--u u) dup IF 0 (um/mod) THEN nip ;

\ mixed maths operators
: MU* ( ud u--ud)  tuck * >r  um*  r> + ;
: T* ( ud u--ut) tuck um*  2swap um*  swap >r 0 d+ r> -rot ;
: T/ ( ut u--ud) dup >r um/mod -rot r> um/mod nip swap ;
: M*/ ( ud u u--ud)  >r t*  r> t/ ;
: UM/ ( ud u--u)  um/mod nip ;
: MU/ ( ud u--ud)  >r 0 r@ um/mod r> swap >r um/ r> ;
: UM*/ ( u u u--ud) >r um* r> mu/ ;

\ double maths operators
: UD* ( ud ud--ud)  dup IF  2swap  THEN  drop mu* ;
: D* ( d d--d)  dup 0< >r dabs 2swap dup 0< >r dabs
                ud*  r> r> xor IF  dnegate  THEN ;
: +D/ ( +d +d--+d)  ?dup IF    dup 1+ 0 1 rot um/  dup >r mu*
      >r over swap r@ um*/  d-
      r> r> swap m*/ nip 0
                          ELSE  mu/ THEN ;
: +D/MOD ( +d +d--+d +d) 4dup +d/  2dup >r >r ud*  d-  r> r> ;
: D0<> ( d--f)  or 0= not ;

\ vulgar simplification...
: GCD ( n n--n)  BEGIN  ?dup  WHILE tuck mod  REPEAT ;
: NORMAL ( v--v)  2dup gcd  tuck / >r  / r> ;

0 1 2 um/ constant HIGHBIT

: UD>U? ( ud--u f) over highbit and or 0= not ;

variable DEN  variable PDEN  variable NUM  variable PNUM

: SETVARS (  ) 0 num !  1 pnum !  1 den !  0 pden ! ;
: NUM>? ( ud--u f) num @ mu*  pnum @ 0 d+  ud>u? ;

\ vulgar simplification continued...
: DEN>? ( ud--u f) den @ mu*  pden @ 0 d+ ud>u? ;
: NEXT! ( u u) den @ pden ! den !  num @ pnum ! num ! ;
: DV>UDV? ( dv--udv f) 2swap dup 0< >r dabs 2swap r> ;
: ?NEGATE ( n f--n) IF negate THEN ;
: REDUCE ( dv--v) [ also ucs ] setvars dv>udv? >r
    BEGIN  2dup d0<> WHILE
           2over +d/mod 2dup num>? WHEN next! drop END
                        -rot den>? WHEN next! END next! 2swap
    AGAIN 2drop 2drop pnum @ r> ?negate pden @ ; previous
```

```
\ vulgar simplification continued
: SMALL ( dv--f) highbit 0 ud< -rot  dabs highbit 0 ud< and ;
: DV>V ( dv--v) 4dup small IF drop nip normal ELSE reduce THEN ;
: V>UV? ( v--uv f) swap dup 0< >r abs swap r> ;
: SNUM>? ( u u--u f) >r num @ * pnum @ + dup r> u> ;
: SDEN>? ( u u--u f) >r den @ * pden @ + dup r> u> ;
: VROUND ( v +n--v) [ also ucs ] >r setvars v>uv? r> swap >r >r
    BEGIN   dup WHILE
     over /mod  dup r@ snum>?  WHEN next! END
     swap r@ sden>?  WHEN next! END
     next! swap
    AGAIN   2drop r> drop pnum @ r> ?negate pden @ ;  previous

\ vulgar arithmetic...
: V* ( v v--v) rot um* 2swap  m*  2swap dv>v ;
: V+ ( v v--v) rot 2dup um* >r >r
               rot m* 2swap m* d+  r> r>  dv>v ;
: VNEGATE  ( v--v)  swap negate swap ;
: RECIPROCAL ( v--v)  swap dup  0< IF  negate vnegate  THEN ;
: V/  ( v v--v)  reciprocal v* ;
: V- ( v v--v)  vnegate v+ ;
: VABS ( v--v) swap abs swap ;

\ vulgar arithmetic continued, vulgar comparison...
1 constant S>V  ( n--v)
: V>S ( v--n)  / ;
: VFRAC ( v--v)  tuck mod swap ;
: VSPLIT ( v--v v) 2dup v>s s>v 2swap vfrac ;
: SIMPLIFY ( v +n--v) >r vsplit r> vround v+ ;
: V0< ( v--f)  drop 0< ;
: V= ( v v--f)  rot =  -rot =  and ;

\ vulgar comparison, vulgar input
: D< ( d d--f)     swap >r >r dup r@ xor highbit and
     r> r> swap rot
     IF  2drop  ELSE  d- THEN  nip 0< ;
: V< ( v v--f)  -rot m*  2swap m*  2swap d< ;
: V0= ( v--f)  drop 0= ;
: VOVERFLOW ( v--f) nip 0= ;
: BASE^ ( +n--+d)  1 0  rot 0 ?DO base @ mu* LOOP ;
: VULGARISE ( d--v) dpl @ base^ dv>v ;

\ vulgar output...
: STRIP ( addr n--addr n)
       BEGIN
          2dup + 1- c@ ascii 0 =
       WHILE 1- REPEAT
       2dup + 1- c@ ascii . = - ;
: >CHAR ( addr ch--n)
       >r 0 BEGIN 1+ swap 1+ tuck c@ r@ = UNTIL
       nip r> drop ;

variable PLACES  10 places !

: SIGN ( n)  -rot sign ; \ My system uses fig SIGN, not '83!
```

the number is too large to be represented, then REDUCE returns 1/0 or -1/0 which can be trapped at any time, as this value will propagate through a running program. Denominator overflow causes a value of zero to be returned, which is represented in a vulgar system as 0/1 or 0/-1. This can also be tested for, although care must be taken not to confuse it with a genuine zero.

It is also possible to detect rounding, as when this occurs the main loop takes a different path than when it does not occur, providing a place where a flag could be set. This would be one possible use for the unused bit in the representation I have chosen, whereby a number could be identified as being completely accurate after a series of calculations, rather than being an approximation. As stripping the bit out before manipulating a vulgar number and putting it back in afterwards would tend to distract from the code, I have not implemented this facility. However, there may be instances where it would be useful, so I mention it.

How It Works

What it does is best shown by a demonstration. Fetch your calculator and follow these steps. Punch up pi, 3.14159 etc. Subtract the integer portion and write it down: 3. Take the reciprocal of the remainder. Subtract the integer portion of this and write it down. Repeat this procedure a few times. You should end up with a sequence of numbers: 3, 7, 15, 1, 292, etc. This is the continued fraction form of pi. Now we use it to generate successive approximations to pi. The first approximation is 3/1 or 3. The second is 22/7, the next 333/106, then 355/113. These successive numerators and denominators are generated by a system similar to

one which produces Fibonacci numbers, with the difference that the current term is multiplied by the next number in the continued fraction before adding to the previous term to produce the next term in the sequence. For numerators, the "zeroth" and "minus oneth" terms are one and zero, respectively, so the sequence goes 0, 1, 22, 333, 355... For denominators, the zeroth and minus oneth terms are zero and one, respectively, so the sequence goes 1, 0, 7, 106, 113...

```
\ vulgar output continued...
: #V ( +v--0 0)  0 swap
    places @ 1+ 0
    DO dup >r um/mod swap base @ um* r> LOOP
    2drop drop
    places @ 0 DO s>d # 2drop LOOP
    ascii . hold s>d #s ;
: V.FR ( v n)      >r over abs swap
      <# #v rot sign #>
      over ascii . >char r> swap - spaces
      strip type ;
: V.F ( v)  0 v.fr  space ;

\ vulgar output continued
: V.SR ( v +n n) >r simplify over abs swap
                 <#  dup 1 = not
                     IF  bl hold
      dup  s>d  #s 2drop  ascii / hold
      2dup  mod s>d  #s 2drop  THEN
                     over 0= >r 2dup < not r> or
                     IF bl hold
                        v>s s>d #s THEN
                     rot sign #>
                  over bl >char r> swap - spaces type ;
: V. ( v) 100 0 v.sr ;
```

In the program, the two phases of producing the continued fraction and summing it are performed concurrently, with the loop ceasing on three conditions. Either the expansion into a continued fraction form has produced a zero, in which case the vulgar fraction has been normalised without overflow, or the numerator or denominator has overflowed, in which case we take the previous approximation, which is the best attainable. The indicators 0/1 and 1/0 derive from the earliest terms in the sequences.

If you want to know why this works, I suggest you read a book about number theory. The key words are continued fractions and convergence. It works, and it produces the best possible approximations. This form of approximation has another advantage which Knuth mentions, although he gives no proof. Using this system, rounding errors tend to cancel out rather than grow with successive calculations. I would very much like to see an explanation of this property.

To speed things up, we wrap REDUCE in a test to see if NORMAL would be sufficient to the task before executing it, as NORMAL is rather faster than REDUCE, giving the word DV>V. I trust the name is self-explanatory.

As REDUCE has three exit points, I have used the Unified Control Structure described in *Forth Dimensions* XIV/6. The word VROUND is structurally similar to REDUCE, and performs a similar function on single-length vulgar numbers, reducing them to fit within a specified range, rather than the width of a cell. This may be used where a loss of precision is an acceptable cost for increased speed of execution. It is also used by some of the numeric output routines, to simplify fractions for legibility purposes.

Simple Vulgar Maths

I am pleased to note that the rest of the code is a lot simpler than this. Indeed, the basic maths operators V* and V+ multiply and add just as we were taught at school, with the difference that they produce intermediate double-length results, which are then rounded down to single-length fractions (i.e., two cells) as described above.

Finding the reciprocal of a vulgar (the result of dividing one by a vulgar) is just a question of SWAPping the numerator and denominator, then moving the sign bit back to the right place. This allows us to divide vulgars almost as simply as we multiply them. Subtraction is equally trivial.

The next few words are concerned with translating single-length integers into vulgars and back again, and extracting the integral and fractional portions from a vulgar number. We will see later that the precision of a vulgar in this representation drops off significantly as the integral part becomes large, so VSPLIT, which separates a vulgar number into an integral-valued vulgar and its proper remainder, is of some use. The integer part is returned as a vulgar fraction to allow them to be recombined later using V+.

SIMPLIFY is similar in function to VROUND, with the difference that attention is only paid to the size of the denominator. This is intended primarily for use in the output routines.

Some Vulgar Comparisons

The comparison words which follow in the listing are mostly trivial. We note that V= in particular expects that the numbers passed to it are in normalised form, so that equal numbers are identical. V< also warrants a brief comment. It is possible that two vulgar numbers could differ by an amount that is smaller than the smallest representable vulgar

Glossary

: 4DUP
(n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2 n3 n4)
The action of 4DUP is as the name suggests.

: D- (d d--d)
Core word. Corrects system fault. See text.

: UD< (ud1 ud2--f)
Returns *true* if unsigned double ud1 is less than unsigned double ud2. Returns *false* otherwise.

: (UM/MOD) (ud ud--ud u)
The recursive portion of UM/MOD (below).

: UM/MOD (ud u--u u)
Core word. Corrects system fault. See text.

: MU* (ud u--ud)
Multiply unsigned double by unsigned single, giving unsigned double result. No overflow check (this is true generally and will not be repeated).

: T* (ud u--ut)
Multiply unsigned double by unsigned single, giving unsigned triple result. Most-significant cell is TOS, as with doubles.

: T/ (ut u--ud)
Divide unsigned triple by unsigned single, giving unsigned double result. Intended for use with T* (above).

: M*/ (ud u1 u2--ud)
Multiply unsigned double ud by unsigned single u1, and divide by unsigned single u2, giving unsigned double result. Uses triple-length intermediate result to avoid overflow.

: UM/ (ud u--u)
Divide unsigned double by unsigned single, giving unsigned single result.

: MU/ (ud u--ud)
Divide unsigned double by unsigned single, giving unsigned double result.

: UM*/ (u1 u2 u3--ud)
Multiply unsigned single u1 by unsigned single u2, and divide by unsigned single u3, giving unsigned double result.

: UD* (ud ud--ud)
Multiply unsigned double by unsigned double, giving unsigned double result.

: D* (d d--d)
Multiply signed double by signed double, giving signed double result.

: +D/ (+d1 +d2-- +d)
Divide positive double +d1 by positive double +d2, giving positive double result.

: +D/MOD (+d1 +d2 -- +d3 +d4)
Divide positive double +d1 by positive double +d2. Returns positive double quotient +d3 and positive double remainder +d4.

: D0<> (d--f)
Returns *true* if double is not equal to zero. Returns *false* otherwise.

: GCD (n1 +n2-- +n)
Returns greatest common divisor of (largest number divisible by) signed single n1 and positive single n2. Returns n1 when n2 is zero, and n2 when n1 is zero.

: NORMAL (v--v)
Reduces an unnormalised single vulgar to its simplest terms. (Both numerator and denominator have their smallest possible values.) Sign bit conventions must be observed in the argument passed to NORMAL.

constant HIGHBIT
Returns a bit pattern. The most significant bit is 1, the rest are 0.

: UD>U? (ud--u f)
Accepts an unsigned double number. If this can be represented as a positive single, returns *true* and its value as a single. Otherwise, returns *false* and an unspecified bit pattern.

variable DEN
Used by REDUCE and VROUND (below). DEN represents the current best approximation to the denominator of a vulgar number.

variable PDEN
Used by REDUCE and VROUND (below). PDEN represents the previous value of DEN.

variable NUM
Used by REDUCE and VROUND (below). NUM represents the current best approximation to the numerator of a vulgar number.

variable PNUM
Used by REDUCE and VROUND (below). PNUM represents the previous value of NUM.

: SETVARS ( )
Used by REDUCE and VROUND (below). Initialises variables used therein. See text for explanation of starting values.

: NUM>? (ud--u f)
Used by REDUCE (below). Returns next approximation to numerator, given next number in continued fraction, expressed as an unsigned double. Flag is *true* if overflow has occurred in numerator, *false* otherwise.

: DEN>? (ud--u f)
Used by REDUCE (below). Returns next approximation to denominator, given next number in continued fraction, expressed as an unsigned double. Flag is *true* if overflow has occurred in denominator, *false* otherwise.

: NEXT! (u1 u2)
Used by REDUCE and VROUND (below). u1 is the next approximation to the numerator of an unsigned vulgar number, u2 is that of the denominator. For both numerator and denominator, NEXT! makes the next approximation the current one, and makes the current one the previous one.

: DV>UDV? (dv--udv f)
Accepts a signed double vulgar number, and returns its absolute value. Flag is *true* if the argument was negative, *false* otherwise.

: ?NEGATE (n f--n)
Returns negation of n if flag is *true*. No action if flag is *false*.

: REDUCE (dv--v)
Returns a normalised single vulgar representing the best approximation to the unnormalised double vulgar passed to it. If the magnitude of the double is too great to be represented, either 1 0 or -1 0 is returned. Numbers too small to be represented are returned as 0 1 or 0 -1 (i.e., rounded to zero). Where a number cannot be represented exactly, it is rounded to the nearest representable value. Numbers that are equidistant from the two nearest values are rounded up or down with equal distribution.

: SMALL (dv--f)
Returns *true* if the double vulgar passed to it can be converted to a single vulgar just by dropping the high cells.

: DV>V (dv--v)
Same action as REDUCE, but works faster when there is a reasonable chance that the double passed to it is small (see SMALL).

: V>UV? (v--uv f)
Returns absolute value of vulgar number. Flag is *true* if vulgar was negative, *false* otherwise.

: SNUM>? (u1 u2--u3 f)
Used by VROUND. Returns next approximation to numerator u3, given next number in continued fraction u1. Flag is *true* if u3 is greater than u2, *false* otherwise.

: SDEN>? (u1 u2--u3 f)
Used by VROUND. Returns next approximation to denominator u3,given next number in continued fraction u1. Flag is *true* if u3 is greater than u2, *false* otherwise.

: VROUND (v +n--v)
Returns an approximation to a vulgar number such that neither the numerator nor the denominator exceed the positive single number on TOS. Comments regards rounding and overflow are the same as for REDUCE.

: V* (v v--v)
Returns the product of the two single vulgar numbers passed to it. Uses a double-length intermediate result, which is then rounded.

: V+ (v v--v)
Returns the sum of the two single vulgar numbers passed to it. Uses a double-length intermediate result, which is then rounded.

: VNEGATE (v--v)
Returns the negation of the vulgar passed to it.

: RECIPROCAL (v--v)
Returns the reciprocal of the vulgar passed to it.

: V/ (v1 v2--v)
Returns a vulgar approximation of vulgar v1 divided by vulgar v2.

: V- (v1 v2--v)
Returns a vulgar approximation of vulgar v2 subtracted from vulgar v1.

: VABS (v--v)
Returns the absolute value of the vulgar passed to it, expressed as a vulgar.

constant S>V (n--v)
Returns the integer passed to it, expressed as a vulgar.

: V>S (v--n)
Returns the integer component of the vulgar passed to it, expressed as an integer. Non-integral numbers are rounded towards negative infinity.

: VFRAC (v--v)
Returns the fractional component of the vulgar passed to it, expressed as a vulgar.

: VSPLIT (v--v1 v2)
Splits a vulgar number into its integral (v1) and fractional (v2) components. Both are expressed as vulgar numbers.

: SIMPLIFY (v +n--v)
Generates an approximation to the vulgar number passed to

it, such that the denominator does not exceed +n. For overflow, etc., see REDUCE.

`: V0< ( v--f)`
Returns *true* if the vulgar passed to it is negative, *false* otherwise.

`: V= ( v v--f)`
Returns *true* if the two vulgars passed to it are identical, *false* otherwise.

`: D< ( d1 d2--f)`
Returns *true* if the signed double d1 is less than the signed double d2, *false* otherwise.

`: V< ( v1 v2--f)`
Returns *true* if the vulgar v1 is less than the vulgar v2, *false* otherwise.

`: V0= ( v--f)`
Returns *true* if the vulgar passed to it is equal to zero.

`: VOVERFLOW ( v--f)`
Returns *true* if the denominator of the vulgar passed to it is zero. This typically happens if a number whose magnitude is too great to be expressed in a rounded form occurs in a word which invokes DV>V.

`: BASE^ ( +n--+d)`
Returns the current value of BASE raised to the power +n, as a double number. +n should be strictly positive.

`: VULGARISE ( d--v)`
Interprets the signed double passed to it as a floating-point number in the current base, and returns a vulgar approximation to that number. The position of the point should be available in DPL.

`: STRIP ( addr n--addr n)`
Strips trailing zeroes from a string. Will not remove a zero that is preceded by a ".". STRIP cannot handle null strings, or strings composed entirely of zeroes.

`: >CHAR ( addr char--n)`
Returns an offset indicating the distance from addr to the first occurrence of the character char. Typically, addr will be the start of a string. char should be present in the string, or >CHAR will search beyond the end of the string.

`variable PLACES`
Holds the number of places after the decimal point which will be displayed when a vulgar number is printed in floating-point format. PLACES is initialised with a default value of 10. It should not be set to zero or less.

`: SIGN ( n)`
Core word. Corrects system fault.

`: #V ( +v--0 0)`
Pictured numeric output word. Converts a positive vulgar fraction, appending it to the output string as a floating-point number, padding out the string with trailing zeroes until there are PLACES characters after the point.

`: V.FR ( v n)`
Prints a vulgar number in floating-point format. The output string is padded with leading spaces until there are at least n characters before the point. Trailing zeroes are stripped out. At least one character must follow the point. No trailing space.

`: V.F ( v)`
Prints a vulgar number as V.FR with the following differences: no leading spaces, one trailing space.

`: V.SR ( v +n n)`
Prints a vulgar number in vulgar format. This consists of an minus sign, if needed, followed by the integral portion of the number, followed by a space. If there is a fractional component, this is printed as numerator/denominator followed by a space. The fraction is rounded using SIMPLIFY with the limit +n. The output string is padded with leading spaces until there are at least n characters before the space separating the integral from the fractional component. Symmetrical division is employed for output, as it is easier to read.

`: V. ( v)`
Prints a vulgar number as V.SR with the following differences: no leading spaces, numerator/denominator will not exceed 100.

number. In this case, subtracting one from the other will give a result that is rounded to zero by DV>V. Therefore, it is not advisable to use subtraction to test for a difference in magnitude. The code given here will detect any difference and is faster than V-. VOVERFLOW is explained above.

Vulgarising Numbers

VULGARISE is probably the most useful word in the suite as, even if you have no great desire to use vulgar arithmetic, every Forth programmer eventually requires good rational approximations to irrational numbers, and this produces such approximations in a format suitable for */, which has been described as the most useful maths operator in Forth. Everyone knows that 355/113 is the best approximation of pi for a 16-bit Forth, but what about a 32-bit Forth—is there a better one?

(Yes! To find out what it is, you will need a 32-bit Forth. Enter the listing, then type:

```
3.14159265358793238 VULGARISE 2CONSTANT PI
```

PI now returns the best rational approximation to pi, which is accurate to 16 decimal places, quite sufficient for most requirements.)

Bear in mind when using VULGARISE that it makes use

of DPL and BASE, which are transitory variables and cannot be relied on to retain their values. The double number passed to VULGARISE naturally cannot have more digits than the Forth will support.

Accuracy and Precision

Finally, having performed our calculations, it may be necessary to output some results, which brings us to the vulgar output words. Vulgar numbers can be output in two forms, as common fractions (i.e., -3 1/2) or as floating point (i.e., -3.5). Both have variable precision, so in the former the fractional part can be simplified to within a specified complexity (i.e., no number larger than 100) and in the latter the number of places after the point (i.e., three decimal places).

Having control over precision invites the question of accuracy and precision in a rational representation. Accuracy and precision are occasionally confused, so let me state what I mean by them. If I said that pi was 22/7, I would be quite accurate but not terribly precise, whereas if I said it was 29.45450752879436751 I would be very precise, but awfully inaccurate. Given the choice between accuracy and precision, I would choose accuracy every time. Regrettably, accuracy is rather harder to quantify than precision.

In the case of a floating-point representation, precision is a matter of how many bits make up the mantissa, and accuracy is a question of how many of those bits can be relied upon. I do not understand the process of determining accuracy in floating point, but gather that even experts tend to rely upon rules of thumb whenever possible in preference to getting embroiled in the complexities involved. This explains the redundancy that is built into floating-point representations in the form of guard bits. One simply bungs in as many guard bits as one hopes will sop up any rounding errors that may occur.

One advantage of floating point in this area is that the gaps between representable numbers are spread through the range such that relative precision remains constant throughout its vast range. Absolute precision falls off towards the high end of the range.

A rational system is a little different. Firstly, it has some redundancy built in, but not in the form of guard bits. In a normalised format, some number combinations, such as 2/4 cannot occur. Specifically, the numerator and denominator must be relatively prime. Determining the number of relatively prime pairs within a range brings us back to number theory again, but a workable approximation is 61 per cent of the possible combinations of two numbers in a range will be relatively prime. This rule of thumb gets better with larger numbers, but we will apply it to our eight-bit Forth. (Remember that?) Ignoring the sign bit for the moment (exactly the same considerations apply to negative numbers, so we need not mention them again), the range of representable numerators is 1 to 127, the same as the range of representable denominators. Therefore, 127*127*0.61=9839 (or thereabouts) rational numbers can be represented. Obviously, if a given fraction can be represented, then so can its reciprocal, so half of these numbers lie in the range 0 to 1, and half in the range 1 to 127. (These are known as proper and improper fractions, respectively.) Within each of these ranges, the representable fractions are distributed quite differently.

Taking improper fractions first: clearly, above 64, only integers can be represented, as 64 1/2 has the representation 129/2, which requires more bits than we have. Equally, above 42 1/3, only multiples of 1/2 can be represented, as 42 2/3 is 128/3. So precision falls off steeply but predictably within this subrange.

Within the proper-fraction subrange, we find ourselves back in number theory land. The distribution of vulgar fractions between 0 and 1 with a limit on the size of the denominator is called a Farey sequence. It has a vaguely fractal look to it, with clusterings of representable fractions within larger clusterings along the number line. It is symmetrical about the midpoint (1/2). The worst possible precision is between 0 and 1/127, and the best is between 1/126 and 1/127, which differ by 1/16002. The average difference between two adjacent numbers between 0 and 1 is 1/(9839/2)=1/4919, approximately. The best approximation to pi is 22/7, which is wrong by about 1/800.

As noted above, accuracy appears to be better than the precision would imply, but this is, at least for me, not proved. My own limited tests, as well as others such as those made by Knuth, tend to uphold the hypothesis.

Other Representations

Other representations are possible, such as a three-cell integer and proper fraction format, which would do wonders for absolute precision, or even an integer exponent and vulgar mantissa, to combine some of the advantages of vulgar fractions with floating point's dynamic range. I have not investigated these areas.

Vulgar Output Words

Now that we have some idea as to how precision goes with vulgar fractions, we can get back to the output routines. We will start off with some utility words. STRIP removes trailing zeroes from a string. It then checks if the last character in the string is a dot and adds one zero back on if it is. STRIP assumes the last character in the string is not a dot.

>CHAR searches for the first instance of a specified character in a string and returns its position. >CHAR assumes the specified character will be present.

The variable PLACES is used to specify how many numbers after the point will be displayed when a number is printed in floating-point format.

#V is the basic vulgar output word. It accepts an unsigned vulgar number and appends the equivalent string to the numeric output buffer. Trailing spaces are included. In common with other "#" words, it returns two zeroes. V.FR demonstrates the use of #V. V.FR strips out trailing zeroes and aligns the number about the decimal point. How many characters and spaces occur before the point is specified by the TOS. V.F is not aligned, but is otherwise similar.

V.SR prints a vulgar number in a simplified vulgar format. It prints the integer portion (with sign if needed), followed by a space and then the fractional part in the familiar numerator/denominator format, if there is a fractional component to the number. In addition to the number to be

printed, it takes two parameters. The TOS specifies alignment, i.e., how many characters and spaces are to be printed before the space that separates the integral and fractional components. 20S specifies the maximum value of the denominator, and hence the complexity of the fractions shown. V. is not aligned, and has a default complexity of 100, but is otherwise similar.

A Comparison With Other Representations

Vulgar fractions are not a replacement for floating-point numbers, but are another alternative amongst a variety of representations for non-integral numbers. They have their own particular strengths and weaknesses. They differ from other representations in that they are not tied to a specific base. Both scaled integer and exponent/mantissa cannot represent accurately numbers that conflict with the binary implementation. This means that numbers such as one-third cannot be expressed exactly. With a rational representation, one-third is 1/3. I suspect this is a factor in its resistance to rounding errors. Another is the fact that the median rounding employed throughout is equally likely to be larger or smaller than the exact value. The rules for predicting which way a rounding will go, up or down, are quite complex.

On the down side, it lacks the dynamic range of floating point, as does scaled integer. Equally, it lacks the years of research that floating point has had applied to it, which is why most worthwhile routines are published for floating point rather than rational numbers. Much as I dislike floating point, this is at times a significant factor. Whilst we are on the subject, there are some very good continued-fraction expansions of most of the transcendental functions. Also, Chebyshev approximation seems ideally suited, as the range of greatest precision, -1 to 1, coincides exactly with Chebyshev's requirements. SQRT is just a question of generating convergents to a continued fraction that can be derived directly and quite simply. And so on.

I have to bluster a little when it comes to the question of speed. I have not been able to do any actual timings, as mine is rendered artificially slow by the high-level UM/MOD. The implementation makes extensive use of UM/MOD. Theoretical timings are complicated, to say the least. The bottleneck lies in the routine DV>V. On one path, it selects Euclid's algorithm and, on the other, a routine that generates convergents to a continued fraction. In fact, this is a variation on Euclid's algorithm, and similar considerations apply to it. Knuth really goes to town on Euclid, and it's all Greek to me.

Unlike floating point, where rounding has no best case, rounding of rational numbers takes a variable amount of time, depending on the numbers presented to it. Although its worst case is slow, most numbers that it receives, if selected randomly, will display best behaviour, or at least very good behaviour, which is significantly faster than worst case. Worst case is where the numerator and denominator are large, and are successive terms in the Fibonacci sequence. Best case happens about 40 per cent of the time, and requires only one trip through the loop.

Therefore, the general impression I get about speed is that it should be a little better than floating point, all other things being equal. Of course, all other things are not equal, but I do not see any reason why one could not design a rational coprocessor. That would, to me, be a nice addition to a Forth processor, effectively adding a new wordset to the processor. (Incidentally, I feel that coprocessors are the way for stack processors to develop, thereby introducing the concept of wordsets—which is central to Forth—to Forth-in-hardware.)

This has been a whistlestop tour through rational numbers. I hope it has whetted your appetite. I finish with a few references.

References

First up comes Knuth, *The Art of Computer Programming: Vol. 2, Seminumerical Algorithms*. He gives three interesting pages about fractions, and twenty-odd, mind-numbingly mathematical pages to Euclid's algorithm. There is other stuff about continued fractions scattered throughout the volume.

Then we have "Numerical Recipes" in *C: The Art of Scientific Computing*, Second edition, by Press, Teukolsky, Vetterling, and Flannery, which is good for stuff concerning the higher functions. They use floating point predominantly, but much could be applied to vulgar fractions.

There has been some work done in Forth. Both myself and Ron Wilson have published related articles in *Forthwrite*, the FIG-UK magazine. Particularly interesting is the second of Wilson's articles, in issue 54, which tackles some of the harder maths. This article supersedes, rather than augments, my earlier work.

The integer maths routines come from my own work in the '91 euroFORML proceedings, and from Prof. Tim Hendtlass' article in *FD* XIV/6, and Prof. Nathaniel Grossman's in *FD* VI/3. Grossman also talks about approximation of vulgar fractions using median rounding.

S.Y. Tang gives an interesting paper, "Approximate Rational Arithmetic" in the '90 FORML proceedings. He gives code for a lot of higher functions, albeit for a representation with a very limited range, making it impractical. I am not convinced that his technique for addition and multiplication is valid, as it reduces one of the arguments to a much simpler form to avoid overflow, rather than reducing a result which gives equal respect to both arguments. Nonetheless, it is a worthwhile paper.

As I am not into heavy maths books, my sources for information about continued fractions, et al., are of an introductory nature. They are *The Higher Arithmetic* by H. Davenport, and *Recreations in the Theory of Numbers* by H. Beiler. The latter book is written by the sort of person who discovers new prime numbers as a hobby, and believes he can write. The section about convergents of a continued fraction starts, "Did you ever dream, perhaps after eating not wisely but too well, of crawling along tortuous passageways and finally emerging into celestial worlds of Maxfield Parrish architecture?" No.

Gordon Charlton is thirty-one years old, and was consequently named after Charlton Heston. He has been described as tall, handsome, witty, intelligent, charming, and world famous. Gordon is also quite tall. He wishes to thank readers of *Forth Dimensions* for not responding to his request for a rigorous description of the Ratcliffe-Obershelp algorithm, as this prompted him to develop a rather better one himself.

The World's Fastest Programmer?

On March 8, 1994, the ACM Special Interest Group on Forth (SIGForth) will invite fifty programmers to joust with computer and compiler, to vie for the title of "World's Fastest Programmer." Individuals and teams will compete to program a physical "gizmo" in the shortest possible time, using the computer and language of their choice.

First held in 1988 under the auspices of the Forth Interest Group, the World's Fastest Programmer contest has travelled from California to Europe, and now arrives in Phoenix, Arizona for the 1994 Symposium on Applied Computing. This symposium is jointly held by six Special Interest Groups of the Association for Computing Machinery:

SIGAPP	(Applied Computing)
SIGAPL	(APL)
SIGBIO	(Biomedical Computing)
SIGCUE	(Computer Uses in Education)
SIGSMALL/PC	(Personal and Small Computers)
SIGFORTH	(Forth)

Now's your chance to prove the worth of your platform, language, or programming methodology! Any computer that supports a parallel printer can be used. Entrants (individuals or teams) need not be members of ACM or SIGForth, but only fifty can compete, so register now!

Registration

The contest registration fee is $25 (U.S.), payable to ACM SIGForth (U.S. checks or money orders only, please). Send your name, address, and a check or money order to the contest chairman:

Brad Rodriguez
Box 77, McMaster University
1280 Main Street West
Hamilton, Ontario L8S 1C0 Canada
e-mail: B.RODRIGUEZ2 on GEnie, or
b.rodriguez2@genie.geis.com on Internet.

(U.S. entrants note: first class postage to Canada is 40 cents!)

For team entrants, only one member need register. You may register by e-mail; however, your registration will not be accepted until the $25 fee is received. Participation is limited to the first fifty registrations received by December 31, 1993. Cancellations after December 31, 1993 will forfeit the registration fee. (The contest organizers reserve the right to extend the registration period at their discretion.)

The contest will be held at the Phoenix Civic Plaza, in conjunction with the 1994 Symposium on Applied Computing (SAC '94) and the 1994 ACM Computer Science Conference (CSC '94). While in Phoenix you may wish to attend SAC '94. For registration information, contact:

Ed Deaton, Conference Director
Department of Computer Science
Hope College
Holland, Michigan 49422 U.S.A.
e-mail: deaton@cs.hope.edu

SAC '94 can also provide information on housing in Phoenix.

Contest Rules

The object of the contest is to solve a real-time programming problem in the shortest time. This problem will involve a hardware "gizmo" to be controlled by a computer.

Rules

1. Entrants may be individuals or teams. Teams may have any number of members.
2. The "gizmo" will be supplied by the contest organizers at the commencement of the contest. Only one gizmo will be supplied to each entrant (i.e., only one gizmo per team).
3. Entrants may use any programming language(s).
4. Entrants may use any computer(s), and any number of computers. Entrants must supply their own computer(s).
5. Entrants must ensure that their computer has an interface suitable for the gizmo, as follows:
   a. The computer must provide 8 bits of parallel output, and 1 bit of parallel input which can be read by software (i.e., not an interrupt input).
   b. The gizmo will use a standard Centronics-type 36-pin female connector, and will electrically resemble a standard "IBM PC compatible" parallel printer. Pins 2 through 9 on this connector (D0–D7) will be the 8 data inputs of the gizmo. The data output of the gizmo will appear simultaneously on pins 11 (BUSY), 12 (PAPER END), and 13 (ON LINE). Ground will be pins 19 through 30. TTL levels will be used. All other pins will be unconnected.
   c. Entrants must supply their own cables. A cable that connects the computer to a standard parallel printer should be satisfactory; however, it is the responsibility of each entrant to verify that the signal assignments given above are compatible with their computer's parallel port, and to provide any necessary adapters.
   d. The gizmo will not require power from the computer.
   e. The contest organizers will exercise care in the design and construction of the gizmo. However, the organizers assume no responsibility for any damage to any entrant's computer(s), caused by connection to the gizmo.
6. No other information about the gizmo will be provided until the start of the contest.
7. No information about the problem to be solved will be provided until the start of the contest.
8. The winner of the contest will be the entrant who completes the assigned problem in the shortest time after the start of the contest. Completion criteria will be provided in the problem description; but no entry will be deemed complete until so pronounced by contest judge(s).
9. The contest organizers may elect to award honorable mentions in other categories, such as shortest program, most readable program, etc. Information about these awards (if any) and their judging criteria will be available immediately before the contest.
10. Entrants must provide their source code to the contest organizers at the conclusion of the contest. Entrants will retain full rights to their work. However, by entering this contest, each entrant agrees to grant the contest sponsors and contest organizers unlimited right to use, publish, or distribute their contest entries. (In particular, the contest organizers intend to publish the winning entry in a suitable journal.)
11. All disputes about the interpretation of these rules, and all other matters pertaining to this contest, will be decided by contest judge(s) to be named by the contest organizers. The decision of the judge(s) will be final.
12. Participation will be limited to the first fifty (50) entrants whose applications are received by 31 December 1993.

A New Forth Development Environment

The Visible Virtual Machine

Ellis D. Cooper, Ph.D.
New York, New York

The Benefit to You

Forth is deep, Forth is fun. This paper is about a new and even more fun way for you to program in Forth. The idea is to automate the chore of visualizing the stack from word to word. I will summarize my experience which motivated the development of this idea. Then I provide a small historical and philosophical argument that the best way to make the Forth virtual machine come alive is to inject "intelligence" into the Forth terminal. The Visible Virtual Machine is described, and some of the plans for enhancements which are in the works. The benefit of this new Forth development environment is the greater programming fun you will have by relegating error-prone chores to the terminal.

1. What are my Forth Credentials?

This is not supposed to be a resume, but I should try to establish my Forth credentials if I am to offer you a new tool for Forth programming. Worshiping in the temple of batch compilation never appealed to me. Interactive languages like BASIC and APL sought my fancy, but walking into the Forth machine shop opened my eyes. In the early 1980's I engineered a hand-held, battery-powered, 80186-based options trading computer running PC/Forth; once I spent several months using NEON on a Macintosh (too bad Chuck Duff's work is not more appreciated, especially his Actor language); I have enjoyed using SC/Forth on the very rapid RTX 2000 for scientific data acquisition; recently I generated a motion-control system using Max-Forth from New Micros, Inc. in their 68HC11; and for a few months I have had to use (the defunct) Bryte-Forth in the 8051 for industrial signal processing.

2. Forth Development in the Past

Right at the outset, being a complete operating system entailed that Forth provide a means to load and edit programs. The choice of input stream (from keyboard or disk) involved the organization of Forth programs into blocks. This was before the widespread use of personal computer word processors, and enabled the target system itself to be the development system with nothing more than a terminal as intermediary between being (the programmer) and object (target system). The next stage in the evolution of Forth development for embedded microcontroller systems put more intelligence into the terminal: it became a PC with a text processor and a download capability to send code to RAM, EPROM emulator, or EPROM programmer. (This dispensed with the artificial segmentation of source programs into blocks.) For example, the New Micros, Inc. system merely requires plugging one of their 68HC11 products onto a serial port. They offer a full-featured communication program called Mirror which has a built-in editor (or the ability to invoke your favorite programmer's editor), the option to toggle "capture to a file," a complete transcript of the development session, and, of course, the protocol for sending a file down to the target. Other than that, everything is the same as it always has been: you get an OK prompt, and you are on your own. What more could you want?

3. Languages, Virtual Machines, and Mental Models

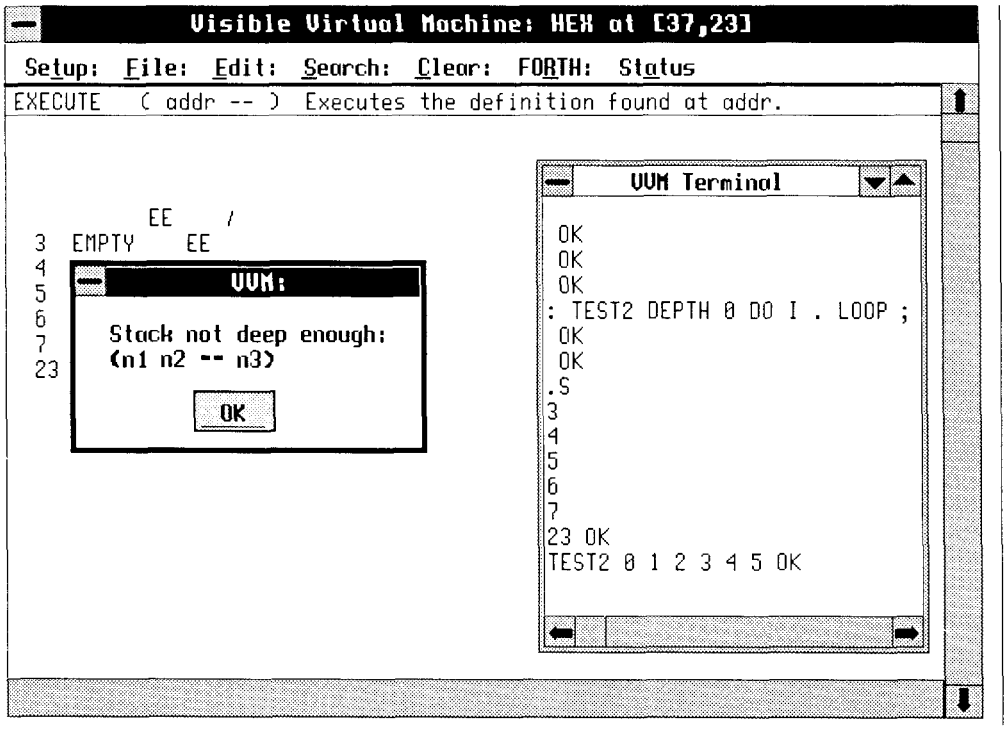
This word "terminal" is a misnomer. It implies something at the end of the line. Nothing is more untrue. A terminal is an intermediary between a being and an object for two-way interaction. In the old days we had "dumb" terminals, and `KEY` and `EMIT` were the sole portals for interaction. Now, we have smarter terminals, but `KEY` and `EMIT` still limit the "bandwidth" of the communication. The competition for survival in the terminal world will be won by "intelligence." This means that the "language" mediating the interaction will evolve towards complexity more tuned to the minds of beings. (No problem with the objects at the other end of the line.)

By making the terminal more complex in the right way, life gets simpler.

Every computer language has a virtual machine, and a person who understands it can tell the story about how the machine works. The brain structure for telling the story is called the mental model. Every Forth programmer starts by understanding the "stack-oriented virtual machine." You

"explain" the Forth machine to someone by telling the story based on your mental model, and he or she "understands" it when they, too, have the mental model. A programmer is always telling stories. He must be able to explain to himself how the program-under-development works. (And exactly the same is true with "he," "himself" replaced by "she," "herself", respectively.)

The point of the paper is that by making the terminal more complex in the right way, with a graphical user interface (GUI), life gets simpler. The GUI for Forth development must visibly show us the virtual machine and let us get our hands right on the "levers" for operating it. This is how KEY and EMIT become wide-open portals for Forth development.



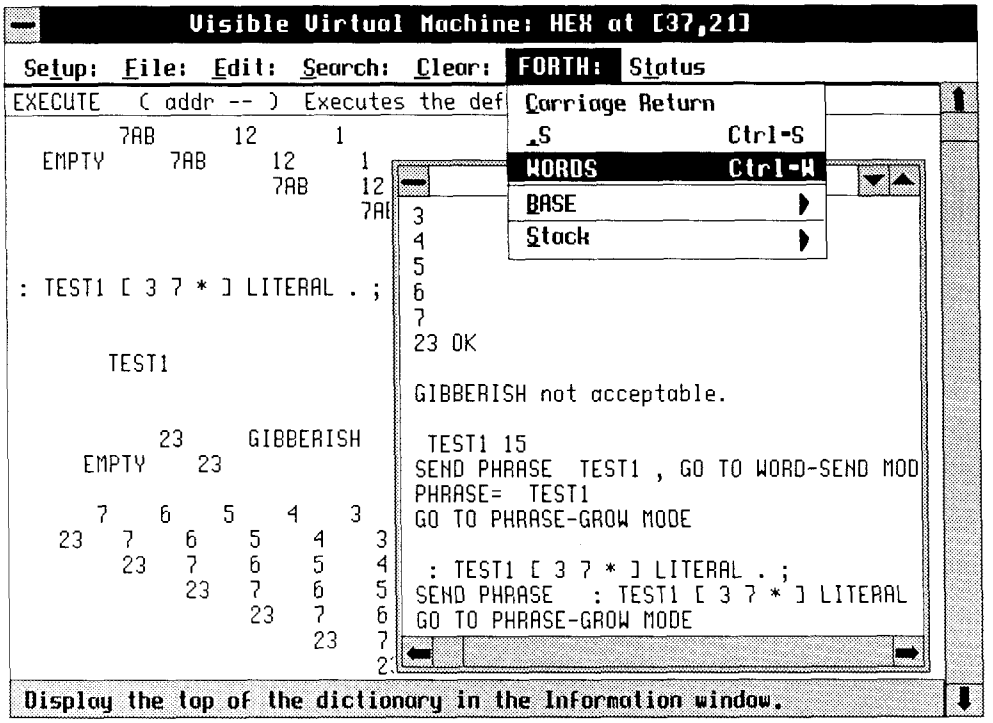
4. The Visible Virtual Machine

I have repeatedly encountered two fundamental problems with Forth. It is undeniably elegant to pass parameters on the stack instead of through named variables. The first problem, though, is the need to develop a mental skill for "keeping the stack straight." If it is more than two or three or four deep, I have to write little columns of stack values between successive words spelled out on a piece of paper. That is the only way I can gain some confidence that everything put on the stack is used, and that every Forth word gets the arguments it needs. This exercise, even on paper, is error prone. The second problem is that one must also develop the skill of imagining the linear organization of the dictionary. It is a real chore to print out a DUMP and then annotate it byte by byte to be sure that some defining word did the right thing. I am sure there are geniuses who never have to do these chores, just like there are people who were born to ski. But when things get tough for me, I try to invent a way out of a boring, error-prone chore.

The basic idea of the Visible Virtual Machine is that you should be able to type a word, press the spacebar, and see the stack as a column of values immediately to the right of the word:

```
   ROT    SWAP    +    XXX       !     DROP
1      3       1    4      7AFE    2
2      1       3    2         4
3      2       2              2
```

More, the chore of checking arity should be automated. "Arity" means the number of arguments on the stack required



by a Forth word, coupled with the number of values left on the stack after the word is executed. The arity of a Forth word is implied by its stack diagram. For example, the stack diagram (n1 n2 -- n3) for + implies arity "2 in, 1 out."

The second idea of the Visible Virtual Machine was that there is no need to develop yet another Forth to do these things. All that is needed is a GUI development language with serial communication capability, for then the Forth of the target system can be installed as a module in the terminal. The bill is filled by Actor 4.03 under Windows 3.1. Years ago, I picked off a bulletin board the Actor code for serial communication and, after a little debugging, it serves perfectly well for talking to embedded systems. In particular, as already mentioned, I use the New Micros, Inc. 68HC11-based products. The entire glossary of Max-Forth provided by NMI was scanned into a text file which is read into the Visible Virtual Machine on startup. The stack diagrams are all there, and the English gloss of word functionality is also instantly available on-screen.

As I said, to fully understand, you need a mental model. To get the mental model of the Visible Virtual Machine (VVM), all you need is to learn the language. The language is all there, in the menu bar of the GUI, which is given below. The VVM supplies two windows. The smaller one in the foreground has two functions. First, it is a standard Forth terminal with the OK prompt, etc. Second, it is used for displaying messages and information generated by the main window. The main window has the menu bar, and is where the virtual machine becomes visible.

The default setting for the stack is that it be shown whenever the spacebar is pressed after typing a word. (This feature can be toggled.) Also, the arity requirements of all words are checked before the word is executed. (This feature can be toggled.) The depth of the stack is checked before putting values on it, to be sure that it does not overflow. (This feature can be toggled.) There are various "hot keys," such as Ctrl-S to see the stack, and Ctrl-W to see the last eight entries in the dictionary.

Both windows are text editor windows, with arrow keys for two-dimensional motion, mouse selection, cut, copy, and paste operations, scroll bars, and paging. The main window is called the "workstrip," and it can be saved as a normal ASCII text file. No longer do you have to re-type a colon definition in your editor after having already entered it in your terminal: just copy and paste. Such files can be loaded or appended to the current workstrip. Pressing the Enter key switches the workstrip into a mode where, instead of sending blank-delimited words immediately, the sequence of words—which is called a "phrase"—is accumulated in a buffer; and when the Enter key is pressed again, the phrase is sent as a whole to the target system. Thus, the Enter key toggles between two modes. A portion of the workstrip selected with the mouse can be commented out with automatic insertion of "(" characters, or can be downloaded directly to the target system. Many other useful capabilities will be added to the VVM, and some of these are briefly described below. The menu bar titles, accompanied by their help bar entries, are displayed in Table One. Included here are actual screen images of the Visible Virtual Machine in action.

5. Forth Development in the Future

Improved visibility would result from being able automatically to see a fully annotated, byte-by-byte listing of the dictionary: show the NFA, LFA, CFA, and PFA with decompiler information for PFA entries and defining word information for the CFA. There should be a search mechanism to find all locations in the dictionary which refer to any specified word, constant, variable, or other defined object. Similarly, the branching vocabulary structure should be visible, and allow cross-vocabulary searches.

It ought to be possible to enter arithmetic formulas with "normal" infix notation and parentheses, and have automatic conversion to efficient Forth stack manipulations. The high-level Actor intermediary provides the additional degree of freedom, so that control structures such as `IF...ELSE...THEN`, `DO...LOOP`, `BEGIN...AGAIN`, etc., could be available at the interpreter level of activity, not only during compilation. Another advantage of the Actor intermediary is that Forth

Table One. Visible Virtual Machine menu bar.

```
=================================================================
"Setup:"
     "COM1"  "Open COM1 serial port."
     "COM2"  "Open COM2 serial port."
    ------------------------------------------------
     "NMI F68HC11 V3.3" "Initialize a target system which uses
                         the F68HC11 V3.3."
     "NMI F68HC11 V3.5" "Initialize a target system which uses
                         the F68HC11 V3.5."
    ------------------------------------------------
    "CLOSE" "Close COMx serial port."
=================================================================
"File:"
     "Load"    "Replace workstrip with file from hard disk."
     "Append"  "Append file from hard disk to workstrip."
     "Save"    "Send selected code of workstrip to target system."
=================================================================
"Edit:"
     "Cut"    "Cut selection to workstrip buffer."
     "Copy"   "Copy selection to workstrip buffer."
     "Paste"  "Replace selection with workstrip buffer contents."
    ------------------------------------------------
     "Clear"   "Delete selection."
     "Comment" "Comment selection."
     "Phrase"  "Copy selection to phrase buffer."
    ------------------------------------------------
    "Select All"  "Select the entire workstrip."
=================================================================
"Search:"
     "Find..."     "Search workstrip for specified text."
     "Find Next"   "Repeat the last search operation."
     "Replace..."  "Replace specified text with new text."
=================================================================
"Clear:"
    "Workstrip" "Clear the entire workstrip."
    "Info"      "Clear the entire information window."
    "Both"      "Clear the workstrip and information window."
    ------------------------------------------------
    "Word"   "Clear the word buffer."
    "Phrase" "Clear the phrase buffer."
    "Stack"  "Clear the stack."
=================================================================
"FORTH:"
    "Carriage Return" "Send a carriage return to the target system."
    ".S"              "Display the parameter stack at the cursor."
    "WORDS"           "Display the top of the dictionary in the
                       information window."
    ------------------------------------------------
"BASE:"
    "DECIMAL" "Send the word DECIMAL to the target system."
    "HEX"     "Send the word HEX to the target system."
    ------------------------------------------------
    "Stack:"
      "Automatic Display" "Automatically display the stack at the
                           cursor after each valid FORTH word."
      "No Display"        "Do not display the stack after each word."
      "Arity Checking"
              "ON"    "Automatically check stack depth prior
                       to executing most recently entered  word."
              "NONE"  "Do not check stack before invoking next word."
=================================================================
"Status"  "Display VVM status in information window."
=================================================================
```

words could be single-stepped through their PFA entries, and even allow stack corrections to be made between steps.

A further feature would reintroduce "blocks" into Forth at a higher level. Namely, there should be a Forth "browser" modeled on the concept familiar to object-oriented programmers. The browser would give access to variable-size blocks of Forth code, and be able to assemble a suitable collection of blocks into a finished ASCII text file, ready to download. All dependencies between Forth words would be resolved automatically.

6. Conclusion

The paper surveys the background, motivation, and features of a new tool which automates some of the chores in Forth programming. The philosophy is to imbue the terminal with ever more "intelligence" in its role as mediator between programmer and target embedded microcontroller system. Future features of the new development environment are described. The Visible Virtual Machine should be immediately useful to Forth programmers. For beginners, it should flatten the learning curve and lighten the load.

I Needed It: Mini-Math

Tim Hendtlass
Hawthorn, Victoria, Australia

Murphy's Laws take many forms—here is one of them: "As soon as you tell somebody you have finished something, you find you haven't." In my case, no sooner had I finished my article, "Math—Who Needs It?" (*Forth Dimensions* XIV/6) than I came across a situation in which I needed a smaller version of the fixed-point arithmetic described there. The application is a neural network which attempts to model some of the behaviour of the human brain. The brain does not provide the equivalent of more than about two or three digits of precision in the signals it uses. The net I am using is large, and the memory requirements and the time taken to move data around were becoming embarrassing. Taking a leaf from the brain, I realised I had a use for a 16-bit number representation which consists of an eight-bit characteristic (the integer bit) and an eight-bit mantissa (the fraction). The more I used them and found how fast and convenient they were, the more I realised they were a natural addition to the number representations I described before.

However, the limited range that can be represented with only eight bits allocated to the signed integer part is a problem at times. So I also wanted a simple way to move between 16-bit fixed-point numbers and "double sized" 32-bit fixed-point numbers (with a 16-bit signed integer part). Being able to convert to 32-bit fixed-point numbers meant I needed words to manipulate them, too. The total package of words to manipulate both 16-bit and 32-bit fixed-point numbers is my Mini-Math pack.

I won't repeat the information about fixed-point numbers I gave before, and refer you to "Math—Who Needs It?" for clarification of anything that follows which is unclear to you. I will refer to a 16-bit fixed-point number (eight-bit characteristic and eight-bit mantissa) as a single fixed number (`SFIX#`), and a 32-bit fixed-point number (16-bit characteristic and 16-bit mantissa) as a double fixed number (`DFIX#`). The 32-bit fixed-point numbers that I described in "Math—Who Needs It?" allowed you to trade off the precision of the characteristic against the precision of the mantissa to best suit your need. The 32-bit fixed-point words given here obtain some speed advantages from fixing these at 16 bits each but, more importantly, give you simple ways to move from single to double fixed numbers, and back again.

You can use the normal integer operators +, -, `abs`, `max`, and `min` on single fixed numbers, just as you would with integers. They "fit" into normal variables and constants, and are stored and read just as you would an integer. The only special routines needed are to perform multiplication (`sfix*`) and division (`sfix/`), and for number input (`sfix#`) and output (`sfix.`).

You can use the normal 32-bit integer operators `d+`, `d-`, `dabs`, `dmax`, and `dmin` on double fixed numbers. They "fit" into normal double variables and double constants, and are stored and read just as you would a double integer.

One or two other convenient features are:

If you divide two integers using the `sfix/` routine (`intA intB sfix/`), the result is correctly expressed as a single fixed number. For example, if you divide 14 by 4 using `sfix/`, the result is 896, which is 3.5 in single fixed point.

Similarly, if you divide one 32-bit integer number by a second 32-bit integer number with `dfix/`, the result is a double fixed number. (`123. 34. dfix/ dfix.` correctly gives 3.6716.)

Indeed, if you take a double fixed number and integer divide it by a single fixed number that has been converted to a double number with the usual integer conversion word s>d, the result after converting by dropping the high word is a single fixed number. (`123. dfix# 34. sfix#`

When I found how fast and convenient they were, I realised they were a natural addition...

`s>d ud/ drop sfix.` gives 3.62.) This latter is only useful if the result is not larger than a single fixed number can represent, otherwise the result is garbage.

The normal word you use to multiply one 16-bit integer by a second 16-bit integer to get a 32-bit answer is `*d` in F-PC. The name might vary for your system. If you multiply two single fixed numbers with `*d` (or its equivalent), the result is a double fixed number. This is very useful if, as I did for example, you have to accumulate the sum of a series of numbers squared and the sum is far too big to express as single fixed numbers.

You can convert any single fixed number into the equivalent double fixed number by converting it to a double integer and double integer multiplying by 256. You can

Tim Hendtlass, Ph.D., is an Associate Professor responsible for the Scientific Instrumentation major at Swinburne Institute of Technology. He discovered Forth in about 1980 and since has used it for research and for teaching to about 80 students per year. In research, he has used it in fields from intelligent adaptive technological support for the elderly, to highly distributed industrial data collection, to devices for the measurement of capacitance under adverse conditions.

Description	Add	Subtract	Multiply	Divide
Single fixed point (16 bits), about two decimal places, written in Forth, portable.	1	1	2.2	2.3
Single fixed point (16 bits), about two decimal places, written in assembler for F-PC.	as above	as above	1.8	1.8
Double fixed point (32 bits), about 4.5 decimal places, written in Forth, portable.	2.4	3.8	13	98–137

convert a double fixed number into a single fixed number by dividing by 256 and dropping the top 16 bits (which should be zero).

The code for the four special words (`sfix*`, `sfix/`, `sfix#`, and `sfix.`) needed to implement single fixed numbers in simple Forth is given below, as are the four special words (`dfix*`, `dfix/`, `dfix#`, and `dfix.`) needed to implement double fixed numbers. I had to find the square root of the sum of all my double fixed numbers, so I implemented a word that takes a double fixed number and returns the square root as a single fixed number. This is included in the listing, in case anyone else finds it useful. It isn't super fast or even super accurate, but I was only working with two or three digits of precision, remember?

A second version for `sfix*` and `sfix/`, and simple words to convert single fixed numbers to and from double fixed numbers, are also given in assembler. The high-level words should (I hope) be usable with any Forth; the assembly words will only work with F-PC. The multiply and divide routines are ones in which the *put-on-the-stack-and-immediately-retrieve-it-from-the-stack-again* overhead of the high-level definitions slows things down substantially, so they benefit most from conversion to assembler.

I give speed figures for the various words implemented in the accompanying code, standardised to a 16-bit integer addition. This allows then to be directly compared with the results for other number representations given in Table Two of my previous article. The 32-bit double fixed divide calculates the answer to the highest accuracy possible (which varies according to the numbers involved), rather than the lesser fixed accuracy of the version from "Math—Who Needs It?" For this reason, the time for the version given here varies and is slightly longer than for that version.

The range for single fixed-point numbers is small, from approximately 127.99 to -127.99, which I have found to be perfectly adequate for most of my purposes. The range for double fixed numbers is from approximately 32767.000 to -32768.000. The ability to painlessly switch up to double fixed numbers when needed, and back down again when the need passes, allows for the most efficient use of resources. If you only need limited precision (a bit better than two or four decimal places) and modest range, and have tight memory requirements, these mini-math words may well suit your purpose.

```
\ A file of 16- and 32-bit fixed-point math operators.
\ Written in 1993 by Tim Hendtlass, Physics Department, Swinburne University
\ of Technology, P.O.Box 218 Hawthorne 3122 Australia
\ phone 63 3 819 8863, fax 63 3 819 0856, email tim@brain.physics.swin.oz.au
\ This is public domain code, use and enjoy. Please let me know of any bugs.
\ Written in, and tested for, F-PC.
\ Except for code versions, should translate easily.

comment:  *************** Single Fixed Number Maths in Forth ***************
SFIX+ same as +, SFIX- same as -.
ABS, MAX, MIN work with single fixed numbers, see text
comment; \ ****************************************************************

: SFIX.
  dup 0< if ascii - else bl then emit \ print sign
  abs split over 253 >                \ fraction > .99?
  if nip 1+ 0 swap then               \ yep, inc integer, set fraction to 0
  (.) type ascii . emit               \ print high byte & decimal point
  10 * split ascii 0 + >r             \ first digit after decimal point
  10 * split swap 128 >               \ need to round up?
  if 1+ dup 10 =                      \ yes now rounded up to 10?
```

```
      if r> 1+ >r drop 0 then          \ if so round up
  then r> emit ascii 0 + emit bl emit \ output digit and one space
;
: SFIX#
  dpl @ 0< if 256                                    \ if integer scale by 256
  else drop 256 1 dpl @ 0 ?do 10 * loop */mod nip    \ scale as approprite
  then
;
: SFIX* ( sfix1 sfix2 -- sfix3 {=sfix1*sfix2} ) 256 */mod nip ;
: SFIX/ ( sfix1 sfix2 -- sfix3 {=sfix1/sfix2} ) 256 swap */mod nip ;

comment: ************* Double Fixed Number Math in Forth ********************
DFIX* and DFIX/ are derived from FIX* and FIX/ in "Math-Who Needs It?"
DFIX+ same as D+, DFIX- same as D-,
DABS, DMAX, DMIN work with double fixed numbers, see text
comment; \ ******************************************************************

: D10* d2* 2dup d2* d2* d+ ;              \ multiply a +ve double number by 10
: DFIX.                                   \ print a double fixed number
  dup 0< if ascii - else bl then emit     \ print sign
  dabs over 65529 u>                      \ fraction > .9999?
  if nip 1+ 0 swap then                   \ yep, inc integer, set fraction to 0
  (.) type ascii . emit                   \ print integer and decimal point
  0 d10* >r                               \ get first digit of mantissa
  0 d10* >r 0 d10* >r 0 d10* >r           \ and second, third and fourth
  32768 u> if 1 else 0 then               \ set up carry if remainder 5 or more
  r> + dup 10 = if drop 0 1 else 0 then   \ apply carry to bottom digit, propagate carry
  r> + dup 10 = if drop 0 1 else 0 then   \ ditto to second bottom, propagate carry
  r> + dup 10 = if drop 0 1 else 0 then   \ ditto to second most significant
  r> + 4 0 do ascii 0 + emit loop         \ do most significant and print digits
;
: T/ ( ut un -- ud )            \ unsigned triple / unsigned single = unsigned double
  >r r@ um/mod swap             \ divisor to r, divide top two 16 bits, rem to top
  rot 0 r@ um/mod swap          \ combine with next 16, divide these by divisor
  rot r> um/mod swap drop       \ repeat for last 16 bits, lose final remainder
  0 2swap swap d+               \ combine parts of answer to form final answer
;
: DFIX#                                 \ enter a double fixed number
  tuck dabs 0 dpl @ 0< if swap          \ save sign, if integer move into correct word
  else -rot                             \ else * 65536
      1 dpl @ 0 ?do 10 * loop t/        \ work out divisor and scaling
  then rot 0< ?dnegate                  \ apply sign as needed
;
: DFIX* ( dfix1 dfix2 -- dfix3 {=dfix1*dfix2})
  rot 2dup xor >r -rot                  \ sign of answer to return stack
  dabs 2swap dabs                       \ make both number positive
  dup>r rot dup>r >r over >r
  >r swap dup>r um* 0 2r> um*
  d+ 2r> um* d+ 2r> * +                 \ assemble 64 bit answer
  rot drop                              \ trim to only 32 bits
  r> ?dnegate                           \ apply sign to answer
;
: T* ( ud un -- ut )           \ unsigned double * unsigned single= unsigned triple
  dup rot um* 2>r um* 0 2r> d+
;
: UD/ ( ud1 ud2 -- ud3 )        \ unsigned double / unsigned double = unsigned double
  dup 0=                        \ top 16 bits of divisor = 0?
  if swap t/                    \ yes, make it a triple and do the division
```

```
  else
      0 over 1 swap 1+ um/mod >r      \ work out scaling factor,copy to return stack
      drop r@ t* drop 2>r             \ scale denominator, move to return stack
      dup 0 2r@ >r t* r> t/ d-        \ calculate (U-U0*W1/W0)
      2r> r> -rot nip >r t* r> t/     \ multiply by (D/W0)
      nip 0                           \ /2^16 (use top 16bits only),make ans double
  then
;
: D/MOD ( ud1 ud2 -- udrem udquot )   \ as UD/ but also gives the remainder
  4dup ud/ 2dup 2>r                   \ do the division, save copy answer
  rot >r over >r >r over >r           \ multiply answer....
  um* 2r> * 2r> * + +                 \                     .....by the divisor
  d- 2r>                              \ calculate remainder, retrieve quotient
;
variable scale
: DFIX/ ( df1 df2 -- dfquot=df1/df2 )
  2 pick over xor >r                  \ work out sign of answer and save
  dabs 2swap dabs 2dup or 0=          \ dividend=0?
  if 2drop r> drop                    \ then make answer=0, lose sign
  else 2swap 2dup or 0=               \ divisor - 0?
    if ." DFIX/ by 0!" abort          \ yep abort
    else 2dup >r >r                   \ nope, keep copy of divisor
      d/mod drop -rot                 \ move integer part of answer below remainder
      0 scale !                       \ initialize scaling
      16 0 do                         \ limit to scaling to no more than 16 bits
        dup $8000 u>=                 \ highest bit of remainder set?
        ?leave                        \ if so, stop the shifting before we overflow
        d2* 1 scale +!                \ no, shift one bit left
      loop r> r> ud/                  \ now divide the scaled remainder
      16 scale @ ?do d2* loop drop swap     \ now unscale the answer, move into place
      r> ?dnegate                           \ put on final sign
    then
  then
;
\ ****************** Square root of a double fixed number ********************
\ Fast divide a double number by 256 while preserving sign.
\ If not using FPC replace with 256. d/
CODE D256/ ( d1 -- d2 )
  pop ax pop dx
  sar ax, # 1 rcr dx, # 1 sar ax, # 1 rcr dx, # 1
  sar ax, # 1 rcr dx, # 1 sar ax, # 1 rcr dx, # 1
  sar ax, # 1 rcr dx, # 1 sar ax, # 1 rcr dx, # 1
  sar ax, # 1 rcr dx, # 1 sar ax, # 1 rcr dx, # 1
  2push
END-CODE
\ algorithm new-guess = old guess +(1-(old-guess)_/target) only valid for numbers > 1
\ As implemented dfix# must be >2 and <16363 and result only good to about 1 in 1000.
: (FSQRT) ( dfix# -- sfix# )          \ find sqrt of dfix# , give answer as a sfix#
  2dup d256/ d2/ drop                 \ first sfix guess=target/2
  begin dup >r dup *d 2over dfix/ d256/ drop    \ (old-guess)_/target
    256 swap - dup abs 2 >            \ correction still worth worrying about?
  while r> +                          \ add on if so
  repeat drop 2drop r>                \ else lose insignificant correction
;
\ add protection for people entering numbers outside the range above. Scale
\ input to >2 and <16 to keep number of iterations to no more than 7.
\ range now 0 to 16274 (scaling slightly reduces high end).
```

```
: FSQRT ( dfix# -- sfix# )
  dup $8000 u> abort" Sqrt of neg number!"  \ abort if negative number
  2dup or 0 =                               \ number zero?
  if drop                                   \ special case, answer = 0
  else 0 -rot dup 2 <                       \ number < 2?
    if begin d2* d2* rot 1+ -rot            \ multiply number by 4
          dup 2 >                           \ we got a number > 1?
       until                                \ loop till we have
    else dup 16 >                           \ got a number > 4?
      if begin d2/ d2/ rot 1- -rot          \ yes, divide by 4
         dup 16 <                           \ we got a number below 16 yet?
         until then
      then (fsqrt)                          \ get square root
      over 0>                               \ need to divide ans by power of two?
      if swap 0 do 2/ loop                  \ do so if required
      else over 0<                          \ need to multiply ans by power of two?
        if swap negate 0 do 2* loop         \ do so if required
        else nip                            \ lose the zero power of 2 factor
        then
      then
    then
;
comment: ****** Single Fixed Multiply and Divide in Assembler for FPC ******
These are a straight translation from the Forth above, minimising the number
of stack pushes and pops by making more use of registers for temporary
storage. Uncomment to use instead of the ones above
*************************************************************************

code SFIX*  ( n1 n2 -- n1*n2 )
  mov bx, # 256 pop ax pop cx imul cx mov cx, bx xor cx, dx 0>=
  if idiv bx 1push then idiv bx or dx, dx 0<>
  if add dx, bx dec ax then 1push
end-code
code SFIX/ ( n1 n2 -- n1/n2 )
  pop bx mov ax, # 256 pop cx imul cx mov cx, bx xor cx, dx 0>=
  if idiv bx 1push then idiv bx or dx, dx      0<>
  if add dx, bx dec ax then 1push
end-code

\ *********** Fast Conversion between single and double fixed numbers *******
code DFIX#>SFIX# ( dfix# -- sfix# )          \ signed /256 and d>s
  pop ax pop dx
  sar ax, # 1 rcr dx, # 1 sar ax, # 1 rcr dx, # 1
  sar ax, # 1 rcr dx, # 1 sar ax, # 1 rcr dx, # 1
  sar ax, # 1 rcr dx, # 1 sar ax, # 1 rcr dx, # 1
  sar ax, # 1 rcr dx, # 1 sar ax, # 1 rcr dx, # 1
  push dx next
end-code
code SFIX#>DFIX# ( sfix# -- dfix# )          \s>d (preserve sign) and * 256
  pop ax cwd xchg dx, ax
  shl dx, # 1 rcl ax, # 1 shl dx, # 1 rcl ax, # 1
  shl dx, # 1 rcl ax, # 1 shl dx, # 1 rcl ax, # 1
  shl dx, # 1 rcl ax, # 1 shl dx, # 1 rcl ax, # 1
  shl dx, # 1 rcl ax, # 1 shl dx, # 1 rcl ax, # 1
  2push
end-code
comment;
```

A Novel Approach to Forth Development Environments for Embedded Real-Time Control

B. Meuris, V. Vande Keere, J. Vandewege
University of Gent, Belgium

This paper describes the features of a novel Forth development environment that has been designed as a global answer, both on the hardware and software level, to the specific needs of real-time embedded control. An investigation is made into general system requirements for real-time distributed control applications. A general design philosophy is procured, providing the fundamental ideas applied in the system design. After a comprehensive description of the system architecture, the paper concentrates on the use of Forth in the development environment. It will be demonstrated that Forth perfectly suits the particular needs of embedded real-time control. A data-flow diagram is presented to define the global system concept. The design and implementation of the major parts of the development environment are discussed in more detail. Finally, conclusions are stated, together with an outline of the current status and future work.

1. Introduction

Forth has often been regarded as a programming language that is very well qualified for embedded real-time control. Taking into account the increased industrial interest in digital control systems, it is appropriate to bring up new ideas about the use of Forth in this application area. A major issue in the design of such control systems is the question of which development environment to use. Faced with the severe requirements that are currently imposed on present-day development environments, one will find only a few Forth development environments, if any at all, which present a satisfactory solution in the application area of embedded real-time control. Moreover, in some cases it may be necessary not only to master the bits and bolts of the final control system, but to know about the internals of the development system. This applies especially to Forth, since the development system actually becomes the final control system. Both act as complementary components in a unified system approach, and are closely linked. Therefore, a profound kowledge of the internal structure of the development system is equally important.

B. Meuris, V. Vande Keere, J. Vandewege. Department for Information Technology (LEA/IMEC), University of Gent, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium. tel.: +32 9 2643316, fax.: +32 9 2643593
E-mail: meuris@lea.rug.ac.be, vdkeere@lea.rug.ac.be

For these reasons, a novel Forth development system has been designed, including both system hardware and software. It is intended for use in the hardware and software development of real-time embedded control systems. An initial analysis of the main system requirements and a subsequent evaluation of the prototype version have guided the current system definition and the specification of a general design philosophy. Some elements of importance are presented next.

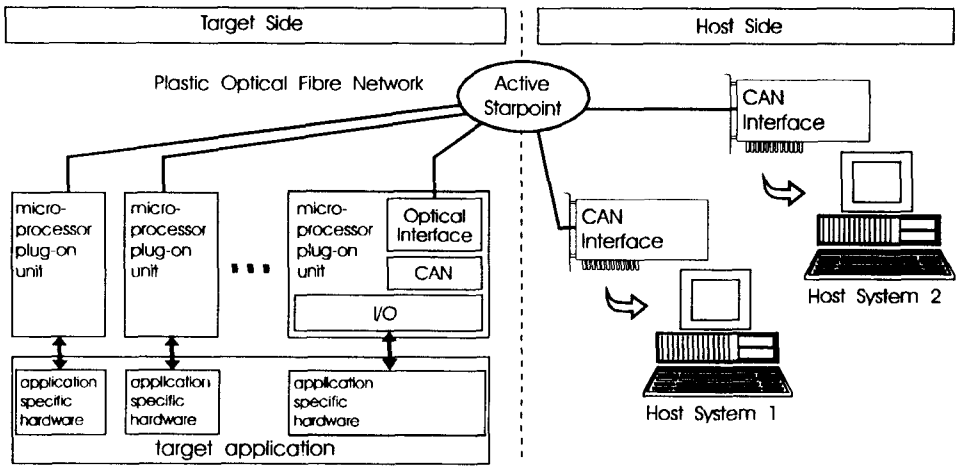
The prime requirement imposed on the system is a distributed architecture, together with common design objectives like high operating speed, flexibility, modularity, low system cost, etc. A distributed architecture implies the use of a communication protocol that manages multiple types of data in an intelligent way. In order to reduce communication bandwidth and, hence, system cost, it is important to have functionality implemented in the right places in the system: focused on embedded real-time control close to the control points, and focused on user interface support close to the programmer. Experience has indicated that a uniform and straightforward user interface is highly appreciated. Furthermore, an open environment in which it is possible to set up all sorts of value-adding tools is preferred. In order to guarantee a smooth transition from design to final use, the entire system is required to simultaneously support development and final operation. Concurrent engineering of heterogeneous components [17] in the final control application is desirable. This requires not only support for simultaneous development of different tasks, but also support for parallel hardware and software design within each task.

The notion of having functions implemented in the right location was adopted as a general system design philosophy, together with special emphasis on well-considered user interface features. Before going into detail about the implementation, an overview of the hardware configuration is given.

2. System Architecture

The hardware configuration of the development system is displayed in Figure One, and is more extensively discussed in [14] and [24]. The backbone of the architecture is a real-time communication network with active star topology. It is implemented optically, using plastic optical fibre (POF) as transmission medium [13]. Different nodes are connected to

Figure One. Hardware configuration of the development system.



a central starpoint. A distinction is made between target nodes and host nodes (i.e., the target and host sides of the system, respectively).

The target side may be composed of one or more individual target units, which are physically part of the control system and execute the control tasks. Each target unit is a combination of an intelligent plug-on unit with additional, application-specific, non-intelligent hardware. Three types of plug-on units have been designed. They span the range of most common performance requirements and are based on microprocessors from the Motorola 68K family. The application-specific, non-intelligent hardware, which is usually not programmable, may consist of transducers (sensors and actuators combining AD and DA convertors), drivers, communication controllers, local displays, ASIC's or dedicated processors (e.g., DSP's), according to the application-dependent requirements. The physical separation of plug-on intelligence and non-intelligent application hardware is justified by the observation that dedicated hardware is, in many cases, significantly more expensive than the microprocessor and its memory. Universal and modular plug-on intelligence offers the benefit of not having to redesign the microprocessor hardware for each new application. Instead, several standard units with different capabilities are available off-the-shelf. The hardware design of the target units for a given application is reduced to just selecting appropriate devices according to the problem specifications and requirements, and interfacing them to the microprocessor on the plug-on unit, mostly by memory mapping. This unit has a Forth system on board, which makes it possible to define an expansion word set for each external device. Another substantial argument in favour of the concept of plug-on intelligence is the fact that it permits more independent and simultaneous hardware and software development, the key element in concurrent engineering.

At the host side, one or more individual host machines provide a means for access to the control system. It is pertinent to note that the function of a target and a host unit are highly distinct. The prime function of a target unit is embedded real-time control, whereas the host is not involved in any control activity at all, and is merely used for interface purposes, both at the development and the operational stage. The general system philosophy is based upon this distinction. A host unit acts as the hardware platform for the Forth development interface while under development, and as the final application interface when the system eventually becomes operational. In order to find an optimal ratio between cost and performance, PC's are used as host units, with MS-Windows™ as the operating system interface.

3. Forth in the Development Environment

A prerequisite to plunging into the software implementation of the development environment, is to carefully consider the use of Forth. A broad diversity of software functionality is required in the system, focused on embedded real-time control at the target side and user interface support at the host side. Requirements are profoundly different in both cases and, by consequence, suggest the use of different programming languages and tools. The consideration of where to use Forth in the development system must be done at an early stage, since it thoroughly determines any further evolution of the design. Other Forth-related issues to reflect on are the question whether the Forth system should be "thin" or "fat," and the choice of a language standard to which the Forth system should comply.

3.1 Strengths and Weaknesses of Forth

The strengths and weaknesses of Forth as a programming language are sufficiently described in literature [6], [10], [11], [15]. Some features are of particular interest in the application field of real-time embedded control. An overview is presented next.

Strengths

As a result of the code threading mechanism, Forth code can be uncommonly compact and requires only few memory resources. Software is developed according to a top-down decomposition, bottom-up implementation paradigm. The implementation stage is characterised by a highly interactive, incremental way of programming, which results in greater debugging ease and increased programmer productivity. The interactive approach allows iterative application of analysis, design, implementation, and validation steps during development. Testing is done close to the point of failure and bugs are more likely to be revealed. The resulting code tends to be safer, more deterministic and, hence, more dependable. Moreover, subsequent validation of small parts of decomposed code usually guarantees more predictable behaviour than single validation of large, undecomposed parts. Programming in Forth consists of extending the language itself towards the specific application and is fairly straightforward, considering the semantic language model without many syntactical constraints. Once the concept of the Forth 'word' as a fundamental language element, and the stack-oriented data processing with reverse Polish notation (RPN), are understood, a programmer quickly gains reasonable programming skills, without having to pass through extensive manuals. Forth programming environments are generally very open and offer direct access to all system levels, including the lowest, hardware-dependent level. As to execution speed, Forth code performs comparably to the code of other compiled languages.

Weaknesses

Forth source code is generally harder to read than the source code of many other programming languages, since it demands more intellectual effort to visualise the action performed by each definition. Furthermore, it is noticed that only a basic functionality is at hand in a more or less standardised form. The availability and/or implementation of more sophisticated features, such as graphical possibilities, file management, complex data structures, mouse control, or windows support, are highly dependent on the language version and often very heterogeneous. This is explained by the fact that Forth's history is marked by several standardisation efforts with varying success. The absence of a universally accepted language standard has led to the existence of many, often vendor-specific, language versions with confusing little differences. These differences do not impose any problems to an experienced Forth programmer, but may startle neophytes and inhibit general language acceptance. Additional weak points can be observed as to operating system and user interface support. Although many Forth systems already run on top of another operating system, little effort is undertaken to keep pace with the evolution of what is commercially available. The adaptation of Forth systems to present-day user interface technologies appears to happen only after considerable delay. As a consequence, it is difficult to find well-trained Forth programmers, and other programmers show little inclination to start using Forth.

3.2 Appropriate Use of Forth

Although Forth has the unique capability of covering all levels of software abstraction [6], it doesn't always provide the best answer to the needs of the programmer. The point is to use Forth in the particular occasions where it outperforms all other programming languages.

Bearing in mind the previously mentioned strengths and weaknesses, it is clear that Forth is well-suited to use at the target side, since the target functionality is focused on real-time embedded control. The applicability of Forth to embedded control has been indicated before in [12]. Basically, it is explained by the fact that the requirements imposed by an embedded application map almost exactly into the strengths of Forth as a programming language. The compactness of Forth code permits operation with a minimum of memory resources. The highly interactive, incremental way of code development has been shown to result in more dependable and predictable code. These features are well-known as paramount for real-time operation [5]. Another aspect of real-time software mentioned in [5] is simplicity, which is sustained by the straightforward programming style of Forth. The low-level hardware access, typical for many control systems, is made possible by Forth in a high-level, hardware-independent way. Also, execution speed and response time to asynchronous events are important for real-time operation, and are well supported by Forth systems.

The main point of interest at the host side is user interface support. Whereas there is little doubt that Forth is well-suited to programming embedded applications, it is questionable whether the same statement also applies to user interface support. A traditional Forth system include a user interface with editor and utilities. This is made possible by Forth's unified approach of closing the entire semantic gap between hardware and application. However, history teaches that Forth was originally designed for hardware control [15], in an era when user interfaces were very limited due to severe technological restrictions.

Although many Forth versions now offer improved user interface support, the weaknesses above still indicate poor suitability for the implementation of user interfaces that are comparable to what is commercially available for other programming languages. It seems that the idea of Forth as a total solution is outdated and is no longer valid, except for systems with limited resources. This observation also applies to operating system support. Although Forth can act as its own operating system, many vendors have adapted Forth to run on top of common operating systems, such as MS-DOS™. Even keeping pace with present-day evolutions in the field of operating system interfaces seems to be equally difficult. Up to now, for example, the authors know about only one Forth programming environment under MS-Windows™ [25]. Given the fact that PC's are preferred as host machines for their optimal cost/performance ratio, and that graphical, window-oriented interfaces tend to become common practice on this type of platform, it is advisable to look for better solutions among languages such as Visual Basic and Visual C++, which feature more advanced possibilities for the implementation of user interfaces.

The previous observations lead to the conclusion that it

is advisable to limit the use of Forth to the target side, and to regard the network not only as a physical separation between host and target units, but also as the border for Forth.

3.3 Forth System Size

The observation that other programming languages are more appropriate to implement user interfaces has immediate repercussions on the size of the Forth system on the target side. All user interface-related functions like source code management and editing, mass storage control, and programming utilities are transferred to the host side for implementation in another programming language. The removal of these functions strips the target Forth system of a substantial number of words and reduces its size significantly. The basic design philosophy, however, goes further. An objective was to concentrate target operation on real-time embedded control, i.e., to enhance system compactness, execution speed, and simplicity: the target Forth system should be a small, fast, and simple processing engine that constitutes the heart of the control system. This implies a "thin" Forth system, containing nothing more than the necessary words for real-time embedded real-time control. Indeed, things like double or floating-point arithmetic, string and block manipulation, search order, and vocabularies are taken to be strictly irrelevant to real-time control, and therefore to be omitted.

It can be concluded that a "thin" Forth system is preferable to a "fat" Forth system. The omitted functionality is either not an absolute requirement (or is even unsuitable) for real-time control, or is better implemented by other means than Forth. As a consequence, the development interface is left with a much more complicated job than just showing things on the screen. Four categories of development interface functions are distinguished: (1) source code editing and management, (2) providing access to the target side of the system, (3) Forth utilities, such as debugging tools, error-handling, help, code pre-processing, and (4) system utilities, like off-line multitasking support or system help. Details on their implementation are discussed in a later paragraph.

3.4 Language Standard Compliance

A last aspect related to the use of Forth in the development system is the choice of the language standard to comply to. Literature clearly shows that standard compliance and standardisation has always been a controversial issue for Forth. [4] refers to the inherent difficulty of standardisation due to the extensibility aspect of Forth. [7] illustrates the problem of standardisation by comparing Forth to a natural language, submitted to natural evolution. However, the authors follow the ANSI Forth Technical Committee currently working on an ANS-Forth, in their recognition of the serious fragmentation of the Forth community and their approach of unification. A certain degree of compliance to a universal Forth standard is necessary for a number of reasons. First, it is easily noticed that it supports certain aspects of real-time software quality, as defined in [8]. Furthermore, [16] mentions a standard as the starting point for language education and for making it easier to bring in trained programmers who may be familiar with it. Standard compliance contributes to unification by presenting a uniform programming platform and encourages the implementation of value-adding tools, such as general C and C++ to Forth translators or application-specific languages like ESTEREL [1] or LACATRE [20], [21], [22], [23] on top of Forth. All previous considerations fit in with the general design philosophy and requirements analysis.

4. Implementation Details

In this section, three parts of the development environment are discussed in more detail, starting with the communication layer which forms the backbone of the system architecture. The implementation of the Forth system at the target side is discussed next. Finally, a description of the development interface at the host side is given.

4.1 The Communication Layer

Versatility and transparency

Gentle exchange of information between the different units in the development environment is an absolute prerequisite for the communication protocol. Moreover, the application domain for which the system has been designed requires well-arbitrated, real-time network operation. By consequence, the choice of network protocol will depend upon its versatility. It must possess the ability to handle the communication related to the development system, as well as the communication related to the final application. It should be noted that the corresponding data may have very distinct characteristics, ranging from short messages that require minimal network access delays (on the order of a few 100 µs.) for extremely time-critical control applications, to large contiguous amounts of data without severe timing constraints for the development environment.

A second requirement in the implementation of the communication layer is uniformity and transparency. The user of the development environment should have access to a well-defined set of communication tools, which provides a means for information exchange between two or more units, without having to be concerned with network arbitration, priorities, error recovery, network management, etc. The ultimate goal is to support transparent communication between any two or more tasks, independent of their physical distribution over the system. In a distributed multitasking development environment, this implies communication between multiple Forths on target units and multiple applications under MS-Windows™.

Implementation with CAN

The network hardware in the development system is implemented optically with an active star topology. Its physical layer uses plastic optical fibre (POF), together with inexpensive LED's and photo diode receivers (770 nm.). It is a low-cost solution, competitive with electric differential twisted pair, but offers a number of specific benefits such as galvanic isolation, compactness, easy handling, robustness, etc. Moreover, a star configuration does not have a single point of failure, in contrast to a ring configuration, for instance [13]. The network protocol implemented on top of this physical layer is the Controller Area Network (or CAN) protocol [3], [18], [19]. CAN is a powerful serial multimaster

protocol, primarily intended for real-time control and multiplex-wiring applications. It originated in the automotive industry and is now finding its way to many applications in industrial control systems. To resolve medium access conflicts, CAN uses non-destructive, bit-wise arbitration. The mechanism guarantees a very short network access latency for highest priority messages (max. 111 µs worst case). The message priority is determined by its frame identifier. Furthermore, CAN has a powerful set of reliability and safety features, including error detection, error signalling, and error recovery by automatic retransmission of corrupted messages. The protocol is implemented in silicon using commercially available CAN controllers.

As a communication protocol, CAN specifies only parts of the physical and data link layer of the OSI reference model [9], viz. physical signalling and the medium access (MAC) sublayer. On top of the MAC sublayer, a CAN network management protocol is being implemented. A central network manager, which can be located either in a host or target unit, supervises communication on the network. In each host or target unit, a local communication manager organises the communication tasks for that unit. At system initialisation, the local communication manager in each unit connected to the network identifies itself to the central network manager. It reports the current status of the unit, its configuration, etc. This information is stored by the central network manager. Through the Forth development interface on a host unit, a programmer can access any target unit in the system. The interface, therefore, calls the local communication manager to set up a communication link to the requested unit. This request is dispatched by the central network manager, which knows the current status of the entire network. According to the available transmission capacity and the priority of other, already existing communication links, the central network manager sets up a new link with a certain priority. This is done by sending the local communication managers at both ends a message to recognise the new link and to allocate memory for data buffering. After setting up and updating the network configuration, the central network manager withdraws and leaves the actual data transmission to the local communication managers at both ends. The above procedure is entirely transparent to the programmer. It is as easy as making a phone call: one dials a number and gets connected by the telephone operator.

All functions to talk to the local communication manager are grouped in a straightforward set of communication tools. At the host side, this set is implemented in a Dynamic Link Library (DLL). DLL services can be made available in almost any programming language that is used to develop applications under MS-Windows™ (C, C++, Visual C++, Visual Basic, Smalltalk...). One application on a host unit using these communication tools is, as explained above, the Forth development interface. Other applications may include, for example, a user interface for the final application. At the target side, the same set of communication tools is present as a Forth word set. It is evident that this uniform and straightforward approach offers many opportunities. A Forth programmer can, for example, in the development interface on a host unit, simultaneously develop and test code for two different tasks on different target units. If these tasks need to exchange data, a communication link can be set up from one target to the other and data transfer can be started by making use of the Forth communication word set on the target unit.

4.2 Forth System Aspects

The importance of compliance to a standard has been demonstrated before. A "thin" Forth system has been shown to be most appropriate. Therefore, the decision was made to implement a ROMmed ANS-Forth system with the minimum required word set, i.e., the core word set. Currently, compliance is established to dpANS-5 [26]. Main implementation options are the 16-bit cell size for highest speed; the availability of a single vocabulary, avoiding the need for search order specification; network-oriented input and output; and a compilation scheme based on macro/subroutine (or M/S) threading. [2] describes this threading scheme and demonstrates it to be optimal for applications where both compactness and execution speed are paramount, as for embedded real-time control. All Forth core words are individually implemented in assembler, which allows them to be optimised for speed and makes it possible to easily integrate existing, native assembler routines into the system. The aspect of limited system portability is, in this case, a minor disadvantage because the development system is based on the idea of universal, intelligent, plug-on units, which are application independent and can be reused without the need for redesign. Error-checking on the target system is deliberately reduced to compromise beween safe, consistent operation and optimal execution speed. The host side, however, provides extensive error-checking tools and debugging features.

Apart from the minimum required core word set, two more Forth word sets are available in the system ROM. They fully depend on this specific environment and are, consequently, not part of any standard. We distinguish (1) a CAN Forth extension word set, representing the well-defined, straightforward communication tools mentioned before, and (2) application-specific Forth extension words, providing a universal means of access to hardware peripherals that are interfaced to the processor. The final control application software is developed on top of these three word sets.

The Forth system on a target can be single- or multi-tasking. The multi-tasking system is based on a scheduler that manages local system resources, such as computing power and memory. Each task runs in a Forth task environment, which is presented to the programmer as if it were a single-tasking Forth system. Hence, the multi-tasking operation is externally invisible. Internally, however, each task locally stores its own contextual user information and word definitions, whereas the underlaying Forth word sets are centralised and shared by all tasks. In other words, a task only requires resources for the storage of information that is necessary for continuing its operation on top of a centralised core functionality when the scheduler allows the task to do so. One important task is the communication manager. Since it must be available on each target, it is incorporated in each Forth system at the assembler level. It provides communication services to input- and output-related words in the core and the CAN extension word set.

The communication itself can be organised in two different ways, depending on whether or not the Forth source code is pre-processed at the host side. In the absence of pre-processing, Forth source code is transmitted as it is entered by the user, i.e., as one long, contiguous sequence of ASCII characters. This form of communication is implementation independent, because the host doesn't need to be concerned about the type of target to which code is sent. Major drawbacks are the large bandwidth requirement and lower communication speed. However, the target Forth system uses execution tokens, which have the size of one cell and uniquely identify each Forth word in the system. Nothing prohibits the translation of parsed words into tokens at the host side instead of the target. In this case, less bandwidth is required, communication gets faster and is more protected, since tokens alone are meaningless without a translation table. A disadvantage is that this kind of communication is implementation dependent, because different Forth systems may use different types of execution tokens.

4.3 Development Interface

Design considerations

According to the general design philosophy, all user interface functions have been transferred to the host side. Programming and monitoring of tasks, source code management, graphical data representation, and providing a final application interface are tasks that are preferably handled in a graphical, windows-oriented environment. It has been argued that these interface functions are better implemented in a programming language designed for this particular job, rather than in Forth.

During the software development phase, a programmer needs a flexible and user-friendly environment in which code can be implemented and validated. It must be tailored to both the specific characteristics of the programming language and the target system it interfaces to. This means optimal support for Forth's interactive and incremental style of software development, good debugging facilities, direct access to the system hardware, and full control of the network communication. A basic functionality should be available standard, such as source code editing and management, and a means for access to the target side of the system. However, it should be possible to extend this functionality by adding extra tools to the environment in a modular way. Examples are Forth utilities such as debugging tools, extensive error handling, programming help, and code preprocessing; and system utilities like off-line multi-tasking support or system help. Hence, an open environment is very much preferable. Finally, it is obvious that the development environment will not run forever, but will be replaced in the end by an application interface. Extra support to smooth this transition should be at hand.

Implementation and features

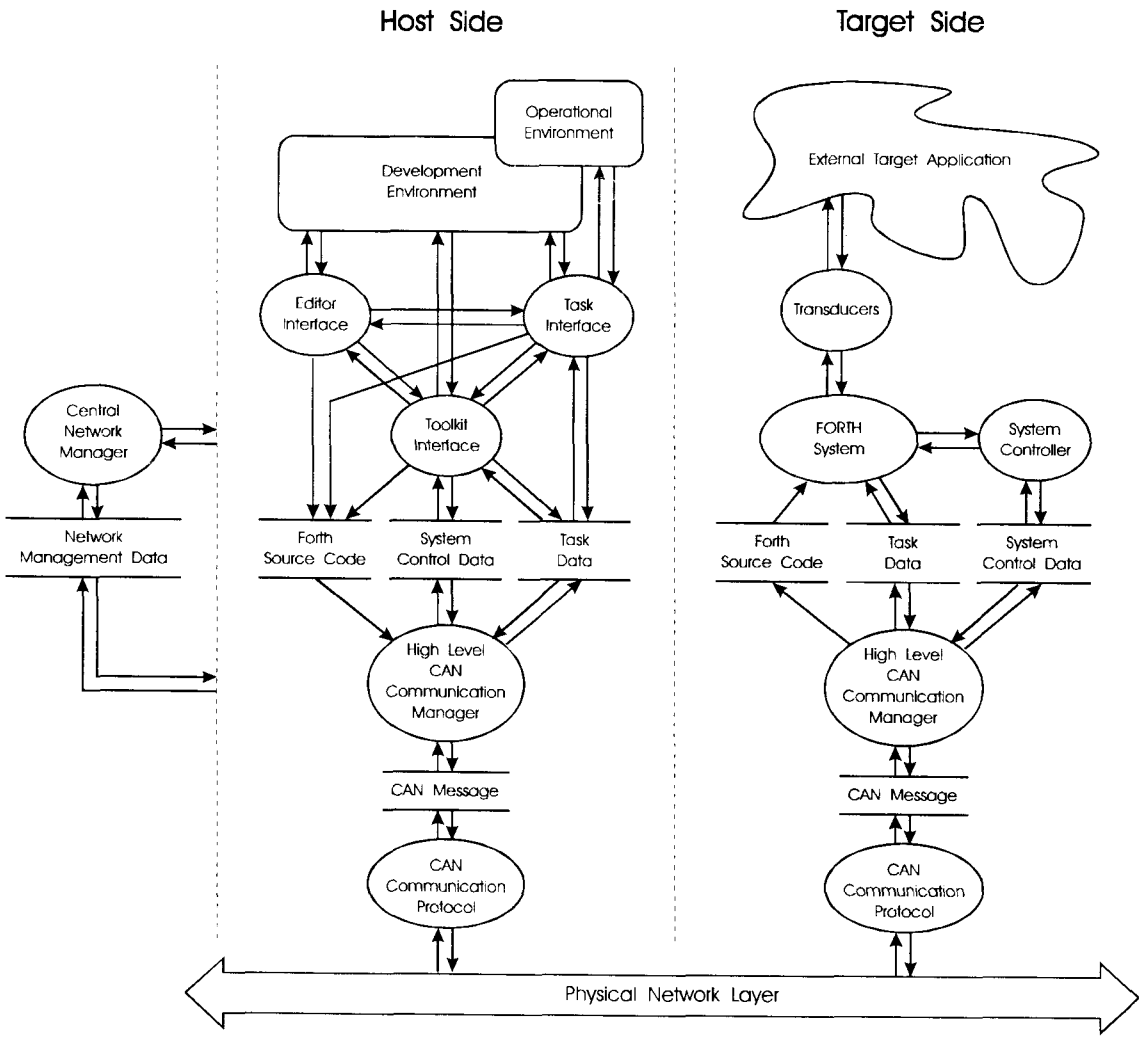
The Forth development interface attempts to formulate an answer to the above needs. A prototype has been implemented under MS-Windows™ in Visual Basic, and the final version is now being developed in Visual C++. Those programming languages have been chosen because they are designed specifically for the development of general applications with the typical MS-Windows™ features, such as a menu bar, a toolbar, cut-and-paste options, etc. The basic functionality of the development interface is centralised around *scripts* and *tasks*, two fundamental concepts associated with source code generation and target access, respectively. Both have highly uniform characteristics.

A script is a set of individual Forth source code files. Brought together in a specific order, they make up a Forth program. A script is comparable to a project, as defined in programming interfaces such as Borland C++. Within a single program, separate files are used to distinguish functionally different blocks of source code, each containing a group of word definitions related to a specific job. In more conventional Forth systems, this distinction is made by using vocabularies. However, the use of vocabularies is somewhat cumbersome in the context of embedded control. Strictly considered, they deal with code management and, therefore, fit in with the host functionality. For this reason, the implementation of vocabularies at the target side has been omitted and translated into the script concept at the host side. Within the Forth development interface, information on the contents of a certain script is presented to the programmer in a script window. Basically, this window gives a list of all files in the script. From the script window, a file can then be opened for editing or file information can be accessed, such as the name of the programmer, the storage format, a brief description, and comments. Multiple scripts can be opened at the same time.

The second concept is that of a task. In the case of multitasking, several programs can run on a single target unit. Each separate program is called a task, and is an independent Forth system that knows the words defined in the program and can be put to action. Within the Forth development interface, the programmer can select tasks from a task list in the task window and assign his programs to them. This selection of tasks is presented to the programmer in exactly the same way as for scripts. The task window contains a list of selected tasks, as well as some essential information on each task, such as the target it runs on, the percentage of allocated memory space that is free, its status, and so on. Each task claims an amount of computing power and memory resources somewhere in the development system. In case a programmer wants to work on-line, he opens the task in the task window instead of assigning a program to it.

Notice again the parallelism between scripts and tasks. For scripts, an editor window opens, presenting the selected file. For tasks, an on-line window opens with the Forth OK prompt, representing the selected task. Furthermore, detailed task information can be accessed, again in the same way as file information for files in a script. The task information window that is opened as a result displays the settings of the allocation of physical resources and configuration options for the selected task. The programmer can use this task information window to change the settings or configuration according to his needs and to make the link between the selected task and the script that will be downloaded to this task. When this information has been entered, downloading a script to a task can easily be accomplished by selecting some download facility in the

Figure Two. Data-flow diagram of the development system.



toolbar. One step further, all scripts can be automatically downloaded to their respective tasks, by selecting some download-all facility. A programmer may not always want to download the whole script, but rather test one or more Forth words that he has written. Therefore, the drag-and-drop feature, common to many MS-Windows™ applications, has been extended to perform a drag-and-drop of selected Forth source code from an editor window to an on-line task window. In this case, a selected fragment of Forth source code is copied from the editor window and attached to the mouse cursor. A simple mouse click drops the code in one of the on-line task windows. This feature has proven to be extremely useful, considering the interactive nature of code development in Forth.

The Forth development interface can further be extended with additional features, which are implemented as individual modules in a toolbox. One of these tools is an interface for the central network management module that supervises network communication. If this module is situated in the host unit where the Forth development interface is running, the programmer will then be able to visualise the current status and capacity allocation of the network, and even change the settings and characteristics if there is authority to do so. Other modules in the toolbox will be related to Forth or the real-time control aspect of the system. Forth tools include syntax checking, error signals, debugging tools like a stack monitor, and context-file generation facilitating code reuse. (Context files are files containing word definitions that are common to many programs in a particular application field. The Forth programmer can select a number of words from a list of all available word definitions ever made, bring them together in a context file, and put that file on the first place in the selected script.) Tools that support real-time software design and multitasking also take advantage of a graphical user interface. Examples are off-line task scheduling and schedulability analysis, prediction of code timing behaviour, memory allocation, and network communication support.

5. Data-Flow Diagram

Figure Two displays a data-flow diagram representing the way in which the development environment and the final application at the host side interact with the external target application at the target side. The possible actions that a programmer can carry out in the development environment take place through either the editor interface, the task interface, or the toolkit interface. These interfaces are linked during development, allowing, for example, code drag-and-drop from an editor to a task window, sending code from an

editor window to a pre-processing tool, or calling a scheduling tool from a task window. For the final application, however, the development environment is replaced by an operational environment which interfaces only to the final system tasks. Each host or target unit has a high-level CAN communication manager which handles all data transfer. These modules distinguish three types of data and interchange them using standard CAN messages on a POF physical network layer. Forth source code goes only down to a target unit, whereas system control data and task data travel bidirectionally. At the target side, a Forth system interfaces to the external target application through appropriate transducers. A system controller module is available for multi-tasking support and Forth system configuration. A central network manager, which can be located either at a host or a target unit, supervises the network communication. It is accessible as a tool on the host side, and controls all the high-level CAN communication managers using special network management data.

6. Conclusions and Future Work

Based on the strengths and weaknesses of Forth, the authors plead for an appropriate use of the language. It has been shown that Forth is well-suited as a language for embedded real-time control. For this application area, a novel Forth development environment has been presented. The general philosophy in this system is to separate real-time and non-real-time features. The development interface is, therefore, implemented in a highly graphical, windowed environment, whereas Forth is used for embedded real-time control through intelligent target units distributed in the system. The distributed concept implies the need for a flexible and uniform communication network. A realisation has been presented in the form of an optical Controller Area Network with an active star topology.

The system has been completely defined at the time of writing this paper. All hardware for the system has been designed and extensively tested. A single-tasking Forth kernel was developed. A first version of the CAN network management protocol and the local communication management software has been implemented, offering basic functionality. Also, a prototype version of the Forth development interface has been written. The authors are currently extending the different parts in the system towards the full system concept as defined in this paper.

7. Acknowledgements

The authors wish to acknowledge in particular Dirk Schamp for his numerous ideas, efforts, and solid patience during the implementation of the Forth development interface. They further wish to express their appreciation and gratitude towards all users of the development system who have helped and inspired them during its implementation.

8. Bibliography

[1] Andre, C. and Peraldi, M. A., "Hard Real-Time System Implementation on a Microcontroller," *Proc. Internat. Workshop on Real-Time Programming*(WRTP '92), IFAC, Pergamon Press, June 1992, pp. 185–189.

[2] Burger, A. and Greene, R., "Comfort: A Faster Forth," *Dr. Dobb's Toolbook of 68000 Programming*, Prentice Hall, New York, 1986.

[3] *CAN Specification* Version 2.0, Philips Semiconductors, 1991.

[4] Colburn, D., "A New Standard for Forth: Bits of History, Words of Advice," *FORML Conference Proceedings*, FIG Inc., San Jose, California, November 1986, pp. 176–180.

[5] Halang, W. A. and Stoyenko, A. D., *Constructing Predictable Real-Time Systems*, Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[6] Harris, K., "The FORTH Philosophy," *Dr. Dobb's Toolbook of Forth*, Volume I, M&T Publishing, Redwood City, California, 1987, pp. 3–10.

[7] Haydon, G. B., "A Forth Standard?" *Forth Dimensions*, Vol. XIII, No. 4, November/December 1986, pp. 28–31.

[8] Hindel, B., "How to Ensure Software Quality for Real Time Systems," *Proc. Internat. Workshop on Real-Time Programming* (WRTP '92), IFAC, Pergamon Press, June 1992, pp. 231–236.

[9] ISO 7498 Information Processing System—Open System Intconnection (OSI)—Basic Reference Model, International Standardisation Organisation, 1984.

[10] Kelly, M. G. and Spies, N., *FORTH, A Text and Reference*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[11] Koopman, P. J. Jr., *Stack Computers, the New Wave*, Ellis Horwood Ltd, Chichester, England, 1989.

[12] Koopman, P. J. Jr., "Embedded Control: Path to Forth Acceptance," *Forth Dimensions*, Vol. XII, No. 1, May-June 1990, pp. 35–37.

[13] Meuris B., Vandewege J., and Demeester P., "Implementation of an Optical Controller Area Network (CAN) using Plastic Optical Fibre," First International Conference On Plastic Optical Fibres and Applications, Paris, France, June 1992.

[14] Meuris B., Vande Keere V., and Vandewege J., "Distributed Real-Time Control Implemented on an Optical Plastic Fibre Star Network," Second International Conference On Plastic Optical Fibres and Applications, The Hague, The Netherlands, June 28–29, 1993.

[15] Moore, C. H., "The Evolution of Forth, an Unusual Language," *BYTE*, Vol. 5, No. 8, August 1980, pp. 76–92.

[16] Peterson, J. V., "The Forth 'Standards'," *FORML Conference Proceedings*, FIG Inc., San Jose, November 1986, pp. 170–175.

[17] Pulli, P. and Heikkinen, M., "Heterogeneous Prototypes in Concurrent Engineering of Embedded Software," *Proc. Internatl Workshop on Real-Time Programming* (WRTP'92), IFAC, Pergamon Press, June 1992, pp. 49–54.

[18] Road Vehicles - Interchange of digital information - Controller Area Network (CAN) for high speed communication. Document ISO/DIS 11898, International Standardization Organization, 1992.

[19] Road Vehicles - Serial data communication for automotive applications. Part 1: Controller Area Network (CAN). Document ISO/DIS 11519 part 1, International Standardization Organization, 1992.

[20] Schwarz, J. J., Miquel, M., and Skubich, J. J., "Graphical Programming for Real-Time Systems. An Experience from the World of Education," *Proc. Internat. Workshop on Real-Time Programming* (WRTP '92), IFAC, Pergamon Press, June 1992, pp. 225-230.

[21] Schwarz, J. J. and Skubich, J. J., "Graphical Programming for Real Time Systems," *Control Eng. Practice*, Vol. I, No. 1, 1993, pp. 43–49.

[22] Schwarz, J. J., Skubich, J. J., Swed, P., and Maranzana, M., "Real Time Multitasking Design with a Graphical Tool," to be published in *Proc. of First IEEE Workshop on Real-Time Applications* (RTAW '93), Manhattan, New York, May 1993.

[23] Skubich, J. J., "Graphical Design of Real-Time Applications," lecture to be published in *Proc. of NATO Advanced Study Institute on Real-Time Computing*, St.-Maarten, Dutch Antilles, October 1992.

[24] Vande Keere, V., Meuris, B., and Vandewege J., "Embedded Real-Time Intelligence in the Physical Layer of Telecom Networks," to be published in *Proc. of First IEEE Workshop on Real-Time Applications* (RTAW '93), Manhattan, New York, May 1993.

[25] WinFORTH by Laboratory Microsystems Inc., USA.

[26] X3J14 Technical Committee, *dpANS-5 draft proposed American National Standard - Programming Language FORTH*, Global Engineering Documents, January 1993.

Comma'd Output for Forth

Charles Curley
Gillette, Wyoming

The addition of commas to numeric text output makes it much more readable. A simple technique to add commas to integer output is shown.

Historical Note

The version of Forth used here is fastForth, a 68000 JSR/BSR-threaded Forth described in Curley, "Optimization Considerations," *Forth Dimensions* XIV/5. There is no code which is specific to fastForth, and implementation on other Forths should be fairly easy. The binary-to-text conversion operators are double precision, in the manner of fig-Forth, 79-Standard, and other common, garden-variety Forths. Being fig-Forth style, variables are initialized at compile time by a value on the stack. `F@` is a fast, word-boundary-only version of `@`. Users of other Forths will have to adjust the code.

Background

Large numbers presented without some sort of internal column device are difficult to read and are liable to errors.

...we add some flexibility to the user's ability to make custom output words...

Anyone who has worked with the Mess-DOS utility CHKDSK on a huge partition is aware that one can easily slip by an order of magnitude in determining the free space on a hard drive. The simple addition of commas (in North America) or periods (elsewhere) to indicate hundreds, thousands, millions, etc., makes for much more readable output. This has long been the custom in financial documents.

The Existing Code

Forth stores all numbers internally as binary data. Binary Coded Decimal (BCD) is not used. Integer values may thus range the full storage capability of the data word (or cell, in dpANS Forthese) on a given implementation.

The typical Forth system for numeric output converts binary values to text by a process of dividing by the contents of the variable `BASE`. Thus, the output string is written into memory a character at a time, from right to left. Typically, a double-precision value is placed on the stack. It may be tested for sign, if signed output is desired.

Conversion is commenced with the word `<#` ("begin sharp"), which initializes the relevant variables. One digit may be converted and added to the string with the word `#` ("sharp"). The value under conversion may be rendered into text until it is exhausted by the word `#S` ("sharps"). Single characters may be added to the string with the word `HOLD`. `SIGN` consumes the single-precision value under the double-precision value being converted to insert a minus sign, if needed, into the string under construction. Conversion is ended with the word `#>` ("end sharp"), which consumes a double-precision value from the stack (presumably, the detritus of conversion), and leaves an address and count ready for `TYPE` or other text output words. The location where the string is built up is system specific, not re-entrant. It may be guaranteed valid only between `<#` and `#>` or shortly thereafter, which encourages the programmer to use the string as quickly as possible.

Conversion to any reasonable number base, and some unreasonable ones, may be done by changing the contents of the variable `BASE`.

To illustrate how this all works, we will examine the word `D.R`, which is used to place signed output of a double-precision value, right-justified in a field of given size:

```
: D.R
\ d fld ---   | type signed d in field of
\               size fld
    >R SWAP  OVER  DABS  <#  #S  SIGN #>
    R> OVER  -  SPACES  TYPE ;
```

`>R` places the field size onto the return stack for later use. The phrase `SWAP OVER` gives us a sign flag on the stack, below the double-precision value for conversion. This sign flag will later be consumed by `SIGN`. We then convert our value to an unsigned value, ready to be converted. Conversion is started with `<#`. The entire value is converted to a string using `#S`, leaving a double-precision value of zero on

the stack. SIGN does its thing, optionally adding a minus sign to the string, and eating the signed value left earlier. We end conversion with #>, which eats the double-precision zero, and leave the address and count of the string on the stack. We now recover the field size from the return stack, subtract the length of the string from it, and print the appropriate number of spaces. Finally, we TYPE the string itself.

Comma'd Output

With this sort of facility available, all we need do is add a variable to hold the delimiter (so the user may change it if he wishes), a variable to count columns, and re-write #S to respect them. In the accompanying code, the variable for tracking columns is PLC ("place counter," or, for the British, "PLC"), and the delimiter is saved in DELIM.

Our version of #S, called #,S ("sharp commas"), begins on line three. We set the place counter to zero. We then begin conversion. The first act of conversion is to convert a digit, leaving a remainder on the stack and a digit in the string buffer. The remainder is then 2DUPed and one copy is ORed together to a single-precision flag for the benefit of WHILE (on line four). If the result of the ORing is zero, the code branches to the return code at the end of line five.

If the result of the test for completion is not zero, we still have at least one more digit to convert. We increment PLC by one, and test for it being equal to three. If it is equal to three, we insert a delimiter character into the string with the phrase DELIM F@ HOLD and then reset the counter to zero. Then it's back to the beginning and another conversion.

That's it, that's the trick part. Everything else consists of duplicating the resident non-comma output functions as needed. D.R (supra) is exactly duplicated with D,.R (line seven), except that we use #,S instead of #S.

```
                                                    Scr #   598
 0 \ comma'd decimal output                    ( 22  3 92 CRC 11:09 )
 1 FORTH DEFINITIONS               BASE F@ >R      HEX
 2 0 VARIABLE PLC                  ASCII , VARIABLE DELIM
 3 : #,S   PLC OFF  BEGIN #  2DUP OR
 4    WHILE  PLC 1+!  PLC F@ 3 =
 5    IF  DELIM F@ HOLD  PLC OFF THEN  REPEAT ;
 6
 7 : D,.R  >R SWAP  OVER  DABS  <#  #,S  SIGN #>
 8         R> OVER  -  SPACES  TYPE ;
 9 : D,.   0 D,.R SPACE ;
10 : ,.R   >R S->D R> D,.R ;
11
12 : ,.    S->D D,. ;      \         : ,?       @ ,. ;
13 \ : W,?    W@ ,. ;                 : C,?      C@ ,. ;
14 R> BASE !
15
```

Design Considerata

In Forth, it is considered good form to eliminate variables whenever possible, but not at the expense of unreadable code. The variable PLC could probably have been eliminated without greatly confusing the code in #,S. However, by making it available to the user, we add some flexibility to the user's ability to make custom output words. Mind you, I haven't the foggiest idea *why* the user might want access to PLC, but he might. Let me know if you figure out a use for it.

Sample Output

The impetus for writing comma'd output words originally was to allow operating system code to present large numbers (such as file sizes and hard drive specifications) in a readable format. This results in the following output for the fastForth version of *df*, a Unix utility which give the user the free space on the disk:

Free Bytes on Drive:	24,070,144
Total Bytes on Drive:	30,751,744
Occupied:	6,681,600
Number of Clusters:	30,031
Sector Size, bytes:	512
Cluster Size, sectors:	2

Much more readable than the typical Mess-DOS or Unix output!

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, lives in Wyoming and has his own private postal code: 82717-2071.

(Letters, continued from page 5.)

b. FIG should make available a conversion/equivalency list. I will tend to keep the entire thing as part of my precompiled system, so I can *always* get results from typing in source code. If this makes you shudder, you just don't get it. My time is *way* more important than 1K. (See, we are *not* talking about embedded systems, so most of your issues just evaporate.)

3. Stop building or optimizing compilers, linking methods, etc., etc. Stop talking about it, too, except in a specialized journal I can avoid. You need to attract a different kind of user, one who values their time and simply wants a good environment from which to solve problems. A good half of the manual of the Forth system I eventually bought ($700) is about issues I couldn't care less about, then it skimps where I need it, with fewer than five pages on the object-oriented stuff or the C-linking package.
4. Vendors: I want to include in my distributed packages the capacity to compile. I am even willing to use a shareware version to get this capability. Don't try to argue me out of it, try to see what you can do. Are my end users (secretaries in law offices) capable of taking your Forth compiler, without manuals or help information, and doing harm to you? When they can get a compiler for free any time they want?
5. There doesn't need to be an issue between screens and ASCII files, sense and support both simultaneously. But give me Ctrl-Q and Ctrl-F, Ctrl-Y and Alt-X out of the package, in both editors.
6. Get out replaceable drivers for the main database packages. Share them, it is one of the most important things you can do for a busy applications developer. Include *indexes.*
7. You could really jump past the other languages if you get the TrueType font support for DOS. How nice of Windows.
8. Even beyond the database managers, there should be the equivalent of dBase itself, complete with all functions and syntax, as a standalone but extensible product. Other programs extol the size of their macro language, but we know that's nothing compared with the power of a Forth. Imagine the performance benefit of a system where you can even select the buffer-management method file by file: single-record use (one buffer), sequential-record use (large buffer, large block read), random with expected frequent reuse (expand to use all available memory, one buffer at a time).
9. Find something new to repeat. I love Forth, but it is clear the same old arguments aren't cutting it. It's kind of like you keep loading the same screen, and the screen keeps aborting with the same message.

Yours,
Jim Mack, President
CoreWorks, Ltd.
126 Front Street, Suite B
Beaver Dam, Wisconsin 53916

[See also, in this issue, sections 3.1 and 3.2 of Meuris, Vande Keere, and Vandewege regarding perceived Forth weaknesses. —Ed.]

(Fast Forthward, continued from page 43.)

performance penalty for the Forth virtual processor. This ignores how efficient the conversion of one syntaxless language to another can become. If Forth were not syntaxless, their fears would be justified.

First, Forth uses a compilation phase to resolve all symbols to executable addresses that the virtual processor can handle. There are no symbol-table lookups to slow the system at run time.

The remaining translation is performed at run time. Mostly, it involves de-referencing one or more levels of pointer indirection, incrementing an instruction pointer, and maintaining the function-return stack. Even the description of this final Forth "translation" process sounds more like the microcode for a JSR instruction rather than a true language-translation effort.

Forth attains the same processor independence and code portability that other high-level languages attain, but with none of the burdens of an intervening syntax.

The C run-time system does roughly equivalent work as it tears down and sets up stack frames for each function that is called.

Now, many Forth systems can directly compile native processor instructions to correspond with high-level Forth source code. With the advent of those systems, Forth performance concerns can be laid to rest, leaving one less perceived problem for Forth. (Other perceived problems are Forth's use of postfix notation and stack operations. But the stack serves as a good intermediary between the virtual Forth processor and the actual hardware. I'd much prefer the debugging of a stack-related problem over the debugging of a problem related to dynamic memory allocation, pointers, or syntax miscues.)

To the extent that other languages hide real processor behavior behind the foreign constructions of syntaxes, we cannot expect them to offer the same positive control that Forth does. The result is unnecessary difficulties creating reliable, bug-free applications.

Considering that it is a high-level language that faithfully steers the underlying computing technology, programmers should be lining up to use Forth!

—Mike Elola

New On-line Chapter

of the Forth Interest Group

Jack J. Woehr
jax@well.sf.ca.us

On-line communications increasingly occupy niches previously reserved for print media or even face-to-face contact. In 1987, the Forth Interest Group recognized this trend in initiating the GEnie Forth Interest Group Roundtable.

Now the Forth Interest Group has authorized its first electronic Chapter: The Whole Earth 'Lectronic Link (WELL) Chapter of the Forth Interest Group.

By the time you read this article, the Chapter will be open for business on the WELL. I'll be your on-line host, and it's my goal to attract enough users to the WELL Chapter to make it worth our collective while to conduct such an experiment.

The WELL is one of America's fastest growing and most enjoyable commercial telecommunication services, and has been mentioned in articles in *TIME* magazine, the *Wall Street Journal*, and elsewhere. The WELL describes itself as a "virtual community" of people from all walks of life. While original participants were mostly computer hackers, the diversity of interests among WELL participants is astounding. The WELL is a friendly site to explore on-line conferencing, the world of the Internet, and the Unix operating system, in addition to being potentially the site of the most dynamic Forth idea exchange on the planet—if you participate!

The WELL is a friendly site to explore on-line conferencing, the world of the Internet, and the Unix operating system.

The WELL is a conferencing system built on top of the UNIX operating system. WELL participants have full access to their favorite UNIX shell: sh, csh, or ksh, along with popular variants thereof, are all available and are included in the basic service, as is ftp, e-mail, and other forms of Internet access. The WELL is truly an open system!

WELL rates and connectivity are excellent also. The WELL costs $15 per month and $2 per on-line hour to subscribe, which includes up to 512 kilobytes of disk storage. Additional disk storage is $20 per megabyte per month, prorated. There is no additional charge for shell access, Internet access, or electronic mail.

To subscribe, you may modem into the WELL directly at 415-332-6106. If that's a long distance call for you, you may access the WELL via Compuserve Packet Network (CPN). In the lower 48 states, the surcharge for CPN access is $4 per hour. Contact the WELL's voice line 415-332-4335, or their fax line 415-332-4927 for more information about CPN access. Alternatively, dial in directly, subscribe, and get the same information on-line for future CPN sessions. Outside the lower 48 states, CPN rates vary. CPN access is currently available from selected nodes on all continents of the planet except Antarctica.

If you are already a user of PCPursuit, the WELL is accessible via PCPursuit San Francisco outdial modems. It's also accessible via other popular intercity outdial services.

Best of all in terms of reducing the phone bill is if you have, as I do, access to a local Internet node at work or at school. In this case, you may rlogin well.sf.ca.us to access the WELL across Internet. Of course, as in all other cases, you must subscribe to the WELL in order to access the WELL. Subscribers accessing the WELL from other Internet sites may also ftp to and from the WELL.

It's our plan to have a full local conference, in addition to providing access to the popular comp.lang.forth on USENET. A large number of Forth files will be copied to the WELL Forth Interest Group Chapter from the RCFB and other sites for download or for ftp'ing (sorry, WELL subscribers only may ftp to and from the WELL). Additionally, since the WELL supports live real-time conferencing, we plan to have on-line Chapter meetings at least once a month.

Subscribe to the WELL by connecting via one of the means described above and then

go fig

to reach the WELL On-line Chapter of the Forth Interest Group.

By the way, do *not* send e-mail to fig@well.sf.ca.us! You see, one of the very first founders of the WELL is named Clifford Figallo (currently of the Electronic Freedom Foundation) and his handle on the WELL is fig. Instead, send e-mail to jax@well.sf.ca.us for more information on the WELL On-line Chapter of the Forth Interest Group.

Note: no portion of your WELL subscription accrues to the Forth Interest Group or to the host of the WELL On-line Chapter of the Forth Interest Group.

Chat you on-line soon on the WELL!

Forth resources & contact information

reSource Listings

Please send updates, corrections, additional listings, and suggestions to the Editor.

Forth Interest Group

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

Forth Interest Group
P.O. Box 2154
Oakland, California 94621
510-89-FORTH (i.e., 510-893-6784)
Fax: 510-535-1295

Board of Directors

John Hall, President
Jack Woehr, Vice-President
Mike Elola, Secretary
Dennis Ruffer, Treasurer
David Petty
Nicholas Solntseff
C.H. Ting

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

In Recognition

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

1979 William Ragsdale
1980 Kim Harris
1981 Dave Kilbridge
1982 Roy Martens
1983 John D. Hall
1984 Robert Reiling
1985 Thea Martin
1986 C.H. Ting
1987 Marlin Ouverson
1988 Dennis Ruffer
1989 Jan Shepherd
1990 Gary Smith
1991 Mike Elola

ANS Forth

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

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

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

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

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

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

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

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

Forth Instruction

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

On-Line Resources

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

GEnie
For information,
call 800-638-9636

- Forth RoundTable *(ForthNet*)*
  Call GEnie local node, then type M710 or FORTH
  SysOps:
  Elliott Chapin (ELLIOTT.C)
  Bob Lee (B-LEE)

- MACH2 RoundTable
  Type M450 or MACH2
  Palo Alto Shipping Company
  SysOp:
  Waymen Askey (D.MILEY)

BIX (ByteNet)
For information,
call 800-227-2983

- Forth Conference
  Access BIX via TymNet, then type j forth
  Type FORTH at the : prompt
  SysOp:
  Phil Wasson (PWASSON)

- LMI Conference
  Type LMI at the : prompt
  LMI products
  Host:
  Ray Duncan (RDUNCAN)

CompuServe
For information,
call 800-848-8990

- Creative Solutions Conf.
  Type !Go FORTH
  SysOps: Don Colburn, Zach Zachariah, Ward McFarland, Jon Bryan, Greg Guerin, John Baxter, John Jeppson

- Computer Language Magazine Conference
  Type !Go CLM
  SysOps: Jim Kyle, Jeff Brenton, Chip Rabinowitz, Regina Starr Ridley

Unix BBS's with forth.conf (ForthNet and reachable via StarLink node 9533 on TymNet and PC-Pursuit node casfa on TeleNet.)*

- WELL Forth conference
  Access WELL via CompuserveNet or 415-332-6106
  Fairwitness:
  Jack Woehr (jax)

- Wetware Forth conference
  415-753-5265
  Fairwitness:
  Gary Smith (gars)

PC Board BBS's devoted to Forth (ForthNet)*

- British Columbia Forth Board
  604-434-5886
  SysOp: Jack Brown

- Grapevine
  501-753-8121 to register
  501-753-6389
  StarLink node 9858
  SysOp: Jim Wenzel

- Real-Time Control Forth Board
  303-278-0364
  StarLink node 2584 on TymNet
  PC-Pursuit node coden on TeleNet
  SysOp: Jack Woehr

Other Forth-specific BBS's

- Laboratory Microsystems, Inc.
  213-306-3530
  StarLink node 9184 on TymNet
  PC-Pursuit node calan on TeleNet
  SysOp: Ray Duncan

- Knowledge-Based Systems Supports Fifth
  409-696-7055

- Druma Forth Board
  512-323-2402
  StarLink node 1306 on TymNet
  SysOps: S. Suresh, James Martin, Anne Moore

- Interface BBS
  Santa Rosa, CA
  707-544-9661
  (ForthNet)* 14.4K BPS
  SysOp: Bob Lee

Non-Forth-specific BBS's with extensive Forth libraries

- DataBit
  Alexandria, VA
  703-719-9648
  PCPursuit node dcwas
  StarLink node 2262
  SysOp: Ken Flower

- The Cave
  San Jose, CA
  408-259-8098
  PCPursuit node casjo
  StarLink node 6450
  SysOp: Roger Lee

International Forth BBS's

- Melbourne FIG Chapter
  (03) 809-1787 in Australia
  61-3-809-1787 international
  SysOp: Lance Collins

- Forth BBS JEDI
  Paris, France
  33 36 43 15 15
  7 data bits, 1 stop, even parity

- Max BBS *(ForthNet*)*
  United Kingdom
  0905 754157
  SysOp: Jon Brooks

- Sky Port *(ForthNet*)*
  United Kingdom
  44-1-294-1006
  SysOp: Andy Brimson

- SweFIG
  Per Alm Sweden
  46-8-71-35751

- NEXUS Servicios de Informacion, S. L.
  Travesera de Dalt, 104-106, Entlo. 4–5
  08024 Barcelona, Spain
  + 34 3 2103355 (voice)
  + 34 3 2147262 (modem)
  SysOps: Jesus Consuegra, Juanma Barranquero
  barran@nexus.nsi.es (preferred)
  barran@nsi.es
  barran (on BIX)

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

ADVERTISERS INDEX

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.
  Hawthorne Public Library
  12700 S. Grevillea Ave.
  Phillip Wasson
  (213) 649-1428

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

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

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

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

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

  Stockton Chapter
  Doug Dillon (209) 931-2448

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

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

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

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

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

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

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

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

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

- **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 Lordi
  (201) 338-9363

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

- **NEW YORK**
  Long Island Chapter
  3rd Thurs., 7:30 p.m.
  Brookhaven National 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**
  Cleveland Chapter
  4th Tues., 7 p.m.
  Chagrin Falls Library
  Gary Bergstrom
  (216) 247-2492

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

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

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

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

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

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

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

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

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

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

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

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

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

INTERNATIONAL

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

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

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

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

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

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

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

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

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

- **GERMANY**
  Germany FIG Chapter
  Heinz Schnitter
  Forth-Gesellschaft e.v.
  Postfach 1110
  D-8044 Unterschleissheim
  (49) (89) 317 3784
  e-mail uucp:
  secretary@forthev.uucp
  Internet:
  secretary@Admin.FORTH-eV.de

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

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

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

SPECIAL GROUPS

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

A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

Mike Elola
San Jose, California

Now Showing—Forth

The Embedded Systems Conference in October was an opportunity to gain exposure for Forth and the Forth Interest Group. I worked in the booth on the first day of the exhibition, along with several other volunteers from the Silicon Valley FIG Chapter. Throughout the three-day show, volunteers distributed literature from FIG and from Forth vendors.

At occasional intervals, a visitor would approach us with a smile indicating that they were aware of Forth. Occasionally, a visitor who knew nothing of Forth would also approach us. All 100 literature kits had been distributed before the end of the third day.

I recall trying to do justice to Forth in one- or two-sentence descriptions. Different booth partners had different approaches. Considering where we were, it seemed appropriate to mention that Forth is well suited to embedded systems development. If our short descriptions of Forth did not win them over on the spot, visitors could read lengthier explanations of Forth in the literature kit.

I encourage builders of bridges between Forth and other development tools.

In time for the show, I had prepared a new, article-length brochure to introduce Forth. Its larger-than-usual format allowed me to elaborate more in support of Forth's claims. The piece still had a marketing purpose, however. It positions Forth in the world of computer languages, identifies its distinguishing features, and highlights Forth benefits that other languages are at a loss to provide.

Jax4th

A new, shareware implementation of Forth made its debut in *Windows NT Developers* magazine. Jax4th is notable for its ANS (draft proposal) compliance. In its first incarnation, Jax4th is less complete than most commercial systems. Kudos to Jack Woehr, the creator of Jax4th, for encouraging Windows developers to become acquainted with Forth.

Deeds that Increase Forth's Appeal

Many programmers will not accept the proposition that Forth is on equal footing with mainstream development tools, while Forth linkers and similar tools are not widely available. So certain utilities can promote Forth simply through their availability. Examples are utilities that help bridge Forth and mainstream development tools.

(A lack of conventional tools such as linkers and library facilities is not evidence of a Forth flaw. Nevertheless, detractors may treat such omissions as though they were evidence of irreversible Forth defects.)

To help make it clear to others that Forth is a high-level language, a contribution such as that of Richard Astle in the preceding issue of *FD* sure helps. I am referring to his article describing how Forth can gain access to precompiled library routines, such as those created using C development tools.

Giving Forth facilities equivalent to those of mainstream programming languages sends the message that Forth is truly a high-level language. Facilities to create and use libraries should be crafted not just once, but over and over again.

I wish to encourage those who are building such bridges between Forth and other development tools. This work bolsters the image of each and every Forth programmer.

If we don't build such bridges, we appear disinterested in what's going on around us. We'll be perceived as "out of touch." As has been suggested by Tyler Sperry, we will be perceived as though we are addicted to substandard tools. (Tyler Sperry is the editor of *Embedded Systems Programming* magazine. He commented upon Forth's future by way of ANS Forth and its acceptance by the Forth community in his September editorial.)

EDN Letter Generates 300 Queries

I got word from the FIG office that 300 inquiries were received in response to a recent letter published by Electronic Design News (*EDN*). In his letter, Tom Napier asserts Forth to be the natural programming language for small microprocessors. He characterizes Forth as a "clean" language, and as the "simplest computer language ever invented."

Syntaxless Languages

While Tyler Sperry thinks there may be a chance for Forth to take up a respectable role in embedded systems development, I am still puzzling over Forth's place in the broader programming realm. Regardless of its area of application, we need to make Forth less of a mystery.

Recently, I latched onto the idea of looking at Forth as a syntaxless language. Perhaps Forth can be marketed as the rare blend of a syntaxless and a high-level language.

Most syntaxless languages are by-products of microprocessor instruction sets. These languages are the definitive low-level languages. (Forth is often dismissed as too low-level a language.) Forth's appeal will be clearer when we understand how syntaxless programming languages can aid programming. I want to show that they offer a better foundation upon which to base a high-level language.

One way that high-level languages distance themselves from processor-level instructions is by introducing a syntax, so that many different behaviors can be invoked through the combinatorial juxtaposition of relatively few keywords and punctuation symbols. Very large swings in meaning can result from changing a single symbol, assuming the syntax rules are still satisfied.

When source code contains misused syntax that still produces syntactically legal code, modern languages will not see it as an error. Instead, the erroneous code is assigned a meaning the programmer was not intending. Errors in syntaxless language seem more conspicuous. Without syntax processing, there is no chance for seemingly reasonable but incorrect expansion or interpretation. Programming languages based on a syntax work hard to make sense out of many different combinations of elements, whether they came together by error or by design.

Different compilers will map different processor instructions to the same source code. Through syntax-based translation, compilers are allowed to synthesize processor instructions according to very complex relationships of language elements, some of which the programmer tends to forget. Accordingly, the translation may lead to the synthesis of instructions that defeat the real intentions of the programmer. Then the programmer must discover which recent code refinement gave the wrong syntax cue to the compiler.

A high-level programming language needs to strike an effective balance between reaching a processor abstraction and preventing us from easily forecasting the actions that will be produced. We don't need to predict the actual processor instructions that are called upon to carry out an action; we do need to be able to identify the type of action that will be elicited.

A better balance may be achieved by omitting syntax altogether. This approach doesn't depend on our ability to think like a compiler with respect to language syntax.

While a processor abstraction is essential for a high-level language, why not choose a "compatible" abstraction? If the processor is syntaxless, the high-level language can also be syntaxless. Forth is a high-level language because it uses an instruction-processing model a level removed from the real instruction-processing resources of the hardware.

Forth uses a thread-execution engine to satisfy the need for processor abstraction. We Forth programmers often refer to the thread-execution engine as a virtual processor (or Forth run-time engine). This reinforces how the Forth language lets the emulation of a processor serve as its processor-abstraction model. Through this means, Forth attains the same processor independence and code portability that other high-level languages attain, but with none of the burdens of an intervening syntax.

At run time, previously compiled code guides a virtual Forth processor (implemented as a handful of low-level routines) through lists of high-level instructions—each of which eventually resolves to a sequence of low-level instructions executed by the real processor. Without any syntax processing to complicate this mapping process, the actual outcome is much more predictable. Upon seeing a trace of the code's execution, the Forth programmer is less likely to receive any rude surprises.

Forth detractors observe that, in effect, some translating occurs at run time. Accordingly, they expect a heavy

(Continues on page 36.)

Product Watch

September 1993

Forth, Inc. announced a new release of its chipForth cross-compiler for targets based on the 68HC11 family. This release sports support for a 32-bit ("big") memory model. It allows the development of lengthy programs for target hardware (theirs or yours) that you communicate with through a PC host. The $1,995 package includes software, target hardware, documentation, and telephone support.

November 1993

Silicon Composers announced the Formula Data Logger32 (FDL32™), a 12 MIPS SBC based on the SC32 Forth microprocessor. It supports up to 64 Mb. of PCMCIA FLASH or SRAM memory, and comes standard with battery-backed 4 Mb. SRAM PCMCIA cards. Two RS-232 and two RS-422 ports are provided along with 13 TTL input lines and ten TTL output lines for parallel I/O. Device drivers are included in source code format.

Companies Mentioned

Forth, Inc.
111 N. Sepulveda Blvd.
Manhattan Beach, California 90266-6847
Phone: 310-372-8493 or 800-55-FORTH
Fax: 310-318-7130

Silicon Composers Inc.
655 W. Evelyn Avenue #7
Mountain View, California 94041
Phone: 415-961-8778

THIS BOOK SHOULD BE IN EVERY FORTH REFERENCE LIBRARY.

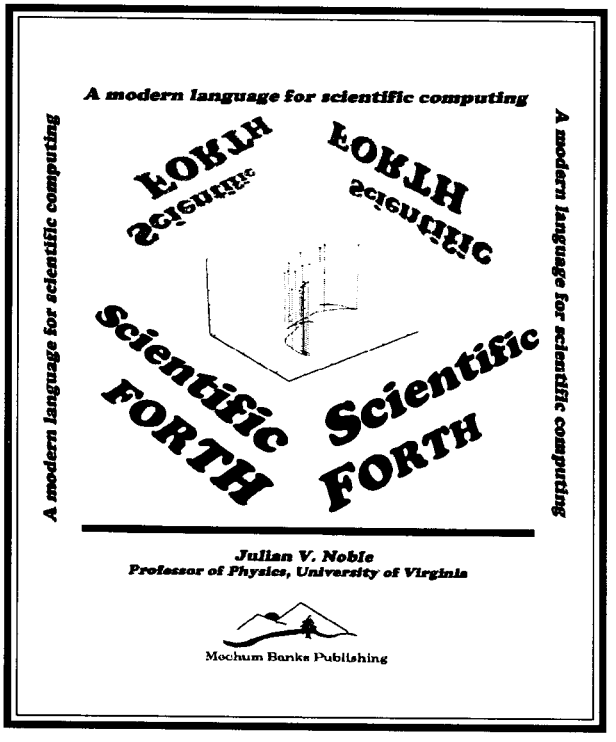
"...FORTH is not usually encountered within the context of scientific or engineering computation, although most users of personal computers or workstations have unwittingly experienced it in one form or another. FORTH has been called "one of the best-kept secrets in computing". It lurks unseen in automatic bank teller machines, computer games, industrial control devices and robots. ...

Some scientists and engineers have gained familiarity with FORTH because it is fast, compact, and easily debugged; and because it simplifies interfacing microprocessors with machines and laboratory equipment....

... FORTH has the ability not only to reproduce all the functionality of FORTRAN —using less memory, compiling much faster and often executing faster also—but to do things that FORTRAN could not accomplish easily or even at all....

One reason FORTH has not yet realized its potential in scientific computing is that scientists and programmers tend to reside in orthogonal communities, so that no one has until now troubled to write the necessary extensions. One aim of this book is to provide such extensions in a form I hope will prove appealing to current FORTRAN users.

Since time and chance happen to everything, even FORTH, I have devoted considerable effort to explaining the algorithms and ideas behind these extensions, as well as their nuts and bolts...."



Scientific Forth
by Julian V. Noble

Scientific Forth extends the Forth kernel in the direction of scientific problem-solving. It llustrates advanced Forth programming techniques with non-trivial applications: computer algebra, roots of equations, differential equations, function minimization, functional representation of data (FFT, polynomials), linear equations and matrices, numerical integration/Monte-Carlo methods, high-speed real and complex floating point arithmetic. (Includes disk with programs and several Utilities)

$50.00