Volume XXI, Number 1,2 May 1999 August

OFFICE NEWS

CONTENTS

DEPARTMENTS

EDITORIAL

Retrospection

I cannot predict Forth's future today any more than I could when I interviewed for this position more than fifteen years ago. Then, fresh from editing *Dr. Dobbs Journal* and in an over-simplification typical of the relatively young, I told the FIG Business Group in Silicon Valley that I believed Forth had not succeeded widely because it had been mismanaged.

That was a sweeping generalization and a poor choice of words, too. I still am a bit surprised they hired me. I was trying to say that the collective energy and vitality of FIG's four or five thousand members (if memory serves) was amazing but wasn't organized or used well enough to promote the language. The energy many other people interpreted as religious-style zeal was mostly turned inward, and reinventing the wheel was a much-favored pasttime. Despite perpetual complaints about the general lack of acceptance of Forth, marketing simply wasn't part of the mindset. Not even a piece in *Rolling Stone*—and how many computer languages can make that claim?—had helped Forth rise much above its grass roots. But inventing and refining a language requires different skills and temperament than marketing it and running an organization.

I see no compelling technical reason now, as I saw none then, why Forth cannot serve as well as any other language and, in enough situations to matter, be the better performer.

For a few years, I've had the opportunity to work with a company where I see evidence daily that Forth has steady work in embedded systems, some amount of general application, and enough mouthwatering projects to keep things exciting. Forth is found everywhere, once you start looking. For that reason, and because Forth embodies some important philosophical aspects of programming, the Forth Interest Group has a purpose.

In my early days at *Forth Dimensions*, after Leo Brodie's departure, the number of readers ensured there usually was more material submitted than pages to print it on. We used a typesetting service, a layout and paste-up artist, and a mailing house. It was high-tech, then, to drive diskettes into town and exchange them for galleys a week later, corrections after that, followed by page proofs and more corrections. When desktop publishing came along, I found it easy enough to design and typeset while I edited; that was good, because the group's size had begun a dwindling process which has continued, although I suspect the rate of decline has decreased. The FIG office changed similarly: a smaller staff with increased efficiency and scope of duties has been brought about by circumstance and enforced by economics.

FIG has done amazingly well, long outliving most special-interest technical groups founded in the nineteen seventies. The techno-culture evolved, and such user groups no longer serve the same purposes, or else they attempt to serve purposes that no longer exist. Perhaps it is time for the Forth Interest Group to reinvent itself.

With some sense of nostalgia, I conclude this, my last editorial for *Forth Dimensions*. I have been unable to continue creating this magazine, in its current form, with the resources and time available to me. With fewer members now, much more editorial time has been required to find material to print. I hope someone will bring fresh perspective, inventiveness, and enthusiasm to the job, and I encourage you to help the Forth Interest Group's administrative staff and its board of directors to provide ways for Forth users to share technical information in a format that is both well designed and compellingly useful.

It has been a pleasure to be associated with Forth—I wish FIG, and each of you, well!

—Marling Ouverson, Editor
editor@forth.org

In Memoriam

Sadly, we learned that Roy Martin died after a long battle with a brain tumor. Roy managed the business affairs of the Forth Interest Group at a time when the organization grew to around five thousand members. He also founded Mountain View Press.

In the early days of FIG, Roy participated wholeheartedly in the FIG Business Group, which directed most of FIG's activities, and he regularly conveyed an inventory of Forth books and *Forth Dimensions* to FIG chapter meetings. His influence helped shape FIG and played no small part in bringing wider attention to the Forth language. He will be missed and remembered, and we offer our sympathies to his family and friends.

Forth Dimensions
Volume XXI, Number 1,2
May 1999 August

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Trace Carter

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 $45 per year ($53 Canada/Mexico, $60 overseas air). For membership, change of address, and to submit items for publication, the address is:

Forth Interest Group
100 Dolores Street, suite 183
Carmel, California 93923
Administrative offices:
831.37.FORTH Fax: 831.373.2845

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. 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 $45/53/60 per year by Forth Interest Group at 1340 Munras Avenue, Suite 314, Monterey CA 93940. Periodicals postage rates paid at Monterey CA and at additional mailing offices.

POSTMASTER: Send address changes to FORTH DIMENSIONS, 100 Dolores Street, Suite 183, Carmel CA 93923-8665.

A Call to Assembly

Introduction

Forth programmers tend to take for granted the assembler that accompanies most Forths [1]. We often eschew assembly language definitions because they are not portable, especially since, in the era of ANS Forth, portability represents an important goal of programming. We therefore resort to the assembler only when running time is of the essence, or when we must access the underlying system at its most basic level—direct control of ports, drives and displays.

However, one of the things members of the Forth community do (besides program in Forth) is attempt to educate their peers who still muddle about with languages of lesser quality. It has for some time seemed to me that in our proselytizing we were missing a good bet, by extolling first the high-level features of Forth—its extensibility, abstractive power, simplicity, elegance, etc., etc. I think it may be better to introduce Forth to the suspicious outsider by way of the Forth assembler.

To make clear why I have taken this position, let me recapitulate what assemblers are and why they exist. In their essence, computer programs consist of sequences of numbers, generally in base 2 (binary) or 16 (hexadecimal) format. Since human brains never evolved to use numbers in any base, the man-machine interface suffered from impedance mismatch in the era when digital computers were programmed directly with plug boards or switches. Programming in this fashion, still common in my youth, was arduous and prone to error.

Fortunately, today's computers rely on specialized conversion programs called assemblers to translate human-readable representations of the instructions in text form ("assembler mnemonics") to their numeric equivalents. Good assemblers recognize macro instructions and operations ("pseudo-ops") that perform such useful chores as referring to variables, constants, or frequently used sequences of instructions by name rather than by address [2].

Even with such tools, however, writing a lengthy program entirely in assembler is not a task to be undertaken lightly. Machine-language programs are hard to get right, hard to understand, and hard to maintain or to port to another machine. High-level languages were invented to provide a better human-computer interface, providing standardized data structures and operations that encompass most of the user's needs, and translating these to machine language in a standardized fashion. Modern optimizing compilers can generate machine code that executes no worse than two times slower than the best hand-coded efforts of wizard hackers. Such facts of life have led to declarations that machine language programming is obsolete [3].

What happens when we encounter problems with no reasonable solution in high-level code? Memory limitations, a desperate need for speed, or an operation trivial at the level of machine registers but time-consuming and circuitous in a high-level language (e.g., bit reversal in Fast Fourier Transform, or interfacing through ports) lead us—however reluctantly—to exercise our constitutional right to assemble.

Most modern programming languages permit linking with assembly language procedures that have been assembled separately (that is, outside the compilation process), thereby combining the ease of high-level programming with the advantages of assembler. The value of this hybrid approach lies in the fact that most programs spend most of their time executing relatively few instructions. Factoring such bottlenecks into separate subroutines, then hand-coding them, can garner large increases in efficiency. The usual procedure is:

1. program, test, and debug everything in high-level code;
2. using a profiler or algorithmic analysis, determine which portions can be rewritten profitably in machine language;
3. finally, endure the tedium attendant on assembling, linking, and testing the hand-coded parts.

In many cases, however, step three is so arduous as to discourage even minimal use of assembly language, except out of desperation.

What we really need is a way to test assembly language subroutines in isolation, i.e., to assemble and run them as separate programs. By eliminating the need to compile an "exercise" program, assemble the subroutine and link the two into an executable, we can telescope the compile–test–debug cycle into a single stage. Once we are satisfied with our machine code subroutines, they can be (re)assembled and linked to the (compiled) main program once, or twice at most.

Forth offers a shortcut that makes assembly language programming as simple as high-level programming. Although Forth is my first choice for the kind of programming I do (numeric and symbolic), not everyone likes it. Moreover, constraints imposed by management often preclude using Forth in commercial applications. However, for assembling and testing isolated machine code fragments—in fact, for rapid prototyping of any sort—Forth is nonpareil and is worth considering for that purpose, even if the final result must be expressed in C or C++.

---

1. That is, all commercial Forths and many public-domain ones.
2. Such assembler directives as macros and pseudo-ops are not actual machine instructions, of course.
3. See, e.g., M. Abrash, *The Zen of Code Optimization* (The Coriolis Group, Inc., Scottsdale, AZ, 1994) for an eloquent defense of assembly language vs. high-level language.
4. L. Brodie, *Starting FORTH, 2nd ed.* (Prentice-Hall, NJ, 1986); *Thinking FORTH* (Prentice-Hall, NJ 1984). M. Kelly and N. Spies, *FORTH: a Text and Reference* (Prentice-Hall, NJ 1986). M. Tracy, *et al.*, *Mastering Forth* (Brady Books, NY 1989). J.V. Noble, *Scientific FORTH: a modern language for scientific programming* (Mechum Banks Publishing, Ivy, VA 1992).

J.V. Noble • jvn@virginia.edu
Charlottesville, Virginia

Julian Noble is among those who display erudition in Forth and who also can writelucidly about it. His work may be enjoyed online (e.g., comp.lang.forth) and, we are pleased to note, in these pages.

Assemblers, cross-assemblers, and decompilers in Forth are so terse that most programmers used to other languages find it hard to believe they are what they claim to be. In a commercial Forth I use regularly, the traditional (postfix) Forth assembler source code resides in a file about 14 Kbytes long, and adds about 6 Kbytes of compiled code to the system; a more elaborate assembler (for a public-domain Forth) that allows prefix style comprises 31 Kb of source and compiles to about 8 Kb; the source file of a generic Forth cross-assembler for Motorola 680x0 CPUs is about 16 Kb; and the assembler for Intel 80486 and Pentium CPUs that comes with a Windows-based Forth is still a relative lightweight at 85 Kb of source. To normalize, the binary of an ancient 16-bit assembler, MASM.EXE (v. 2.0), is about 74 Kb long.

The Forth assembler is written in Forth, hence it operates the same way as any other set of Forth words. The words for compiling a new definition from assembler mnemonics, analogous to `:` and `;`, are `CODE` and `END-CODE`. Rather than threading together the addresses of predefined words from the dictionary, the assembler mnemonics actually assemble a new machine code fragment containing the opcodes of the target CPU. For the sake of definiteness, I shall illustrate with popular public-domain Forths F-PC and Win32Forth (its lineal descendent), both the brain-children of Tom Zimmer and readily available from the Web site www.taygeta.com.

The pedagogical advantage of a complete F-PC or Win32Forth installations is that they provides access to the machine code of the most primitive kernel words. These serve as convenient examples of the assembler's operation, as well as of how to program simple operations in Intel 80x86 assembler.

This note provides three examples of the development process: `STIB` [5], a routine that bit-reverses numbers (for use with Fast Fourier Transform); `UCASE`, a routine to convert all lower-case letters in an ASCII string to upper case, leaving digits and punctuation alone; and `SPHBES`, a spherical Bessel function. In what follows, we assume the reader is familiar with the assembly language mnemonics of the Intel 80x86 series of CPUs. Occasionally their operation will be amplified in detail; however the reader is advised to consult a standard assembly language programming manual [6].

Bit-reversal

The bit-reversal routine `STIB` may be written in high-level Forth [as in Listing One].

How does this work? The subroutine expects an integer *n* on the stack, in the range

$$0 < n < 2^k = N,$$

where *N* is the order of the FFT (power of 2). The loop must be executed $k = \log_2(N)$ times, so the loop limits are 0 and *k*. For simplicity, *k* is placed on the stack above *n*, rather than fetched from a variable. To see how the routine performs bit reversal, visualize the (input) integer *n* in binary notation: a string of 1's and 0's in a field *k* bits wide. For example, if the order *N* of the FFT is 16 then the field is *k*=4 bits wide; the number 7, e.g., is represented as

n = 7d = 0111b,

and its bit-reversed form is

n′ = 1110b = 14d.

We start with *n*-=0 (all bits are 0); we then shift *n*′ one position to the left, adding to it the right-most bit of *n*. Then we shift *n* one position to the right (with its former right-most bit dropping into oblivion), and then repeat until done.

We simulate the shift operations using integer divide-by-two (`2/`) for the right-shift, and multiply-by-two (`2*`) for the left-shift. We keep *n* and *n*′ on the data stack (equivalent to temporary local variables that are reclaimed when the subroutine returns control to the main program).

Testing immediately, as is our wont in Forth,

```
 4 7 STIB . <cr> 14 ok
14 7 STIB . <cr> 7 ok
```

A machine code version that carries out the operations entirely within the CPU's registers will execute much faster than the high-level code [7]. The logical right-shift (`SHR`) and rotate-left-through-carry (`RCL`) instructions are key to an exceedingly simple subroutine. Their behaviors are illustrated by the figure below.

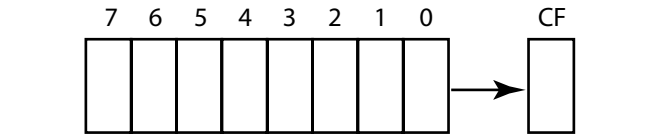
5. That is, "BITS" spelled backwards. Forth names often seem odd to programmers used to the baroque compound names of C functions. Forth's conventions aim toward self-documenting code, with telegraphic word names that express their functionality without lengthy marginal notes. `BIT_REV` would also work, and may perhaps be less cryptic.
6. See, e.g., L. Scanlon, *Assembly Language Programming for the IBM PC AT* (Prentice Hall, New York, 1986); or J.H. Crawford and P.P. Gelsinger, *Programming the 80386* (SYBEX, Alameda, CA, 1987).
7. The speed increase is large—at least 20- or 30-fold in standard Forth, which does not compile to an optimized machine-code image. Comparison with the same function written using an optimizing C compiler, say, may reflect only a two- or three-fold increase in speed.

Listing One

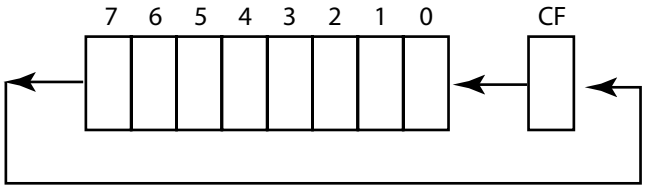
```
: STIB ( k n — n')           \ reverse order of bits
    0 SWAP ( — k 0 n )       \ initialize n' ( — lg2[N]  n' n)
    ROT   0 DO               \ loop k times
              DUP  1 AND     \ pick out 1's bit of n
              ROT  2*  +     \ leftshift n' 1 place, add 1's bit
              SWAP  2/       \ rightshift n 1 place
          LOOP  DROP         \ end loop, discard n
;                            \ end definition
```

Listing Two

```
                        \ initialization steps
     POP BX             \ obtain n
     XOR DX, DX         \ n' = 0
                        \ repeat following instructions k times
     SHR BX, 1          \ logical right shift 1 place
     RCL DX, 1          \ rotate left through carry 1 place
```



SHR



RCL

Different Forths will require minor differences in how we proceed. Several commercial Forths cache the top of the data stack in the register BX, thereby eliminating some pushes and pops. The public-domain F-PC, on the other hand, leaves BX free. Since we are illustrating with F-PC, our first job will be to obtain the argument *n*; we therefore `BX` it from the stack to `BX`:

```
POP BX
```

Next let us assign the (unused) `DX` register to the bit-reversed answer; we initialize `DX` to 0 quickly using bitwise exclusive-or [8]

```
XOR DX, DX
```

Now we shift `BX` one place to the right using `SHR`; the rightmost bit, as the Figure suggests, moves from the register to the Carry Flag. Then we `RCL` the `DX` register one place to the left; the bit formerly in the Carry Flag becomes the rightmost bit of `DX`. The left-most bit (if any) of `DX` ends up in the CF. But that does not matter, because it will be replaced by the right-most bit of `BX` when the sequence is repeated. So the machine-language program (with comments) looks like [Listing Two].

8. The instruction `MOV DX, # 0` would also work, but requires one byte more storage.

All that is required now is to arrange to repeat the two-instruction sequence the requisite number of times. For simplicity, let us do this using the most elementary looping instruction, `LOOP`. We must place the number of times the loop is to be executed in the register `CX`, then at the end of the loop issue the `LOOP` instruction which will decrement `CX` by 1 and loop back to the starting point (which we must label somehow—we will return to this point and describe how it is done), as long as `CX` is non-zero. That is, it will loop the number of times specified by the integer in `CX`.

To assemble this subroutine using an assembler like MASM® or TASM®, we would prepare a text file of the form:

```
  POP BX          ; get n
  POP CX          ; get # of iterations
  XOR DX, DX      ; set n' = 0
HERE:             ; beginning of loop
  SHR BX, 1       ; send 0'th bit of n to CF
                  ;    and shift right 1 place
  RCL DX, 1       ; shift n' left and
                  ;   move CF into 0th bit of n'
  LOOP HERE       ; CX=CX-1, loop if CX > 0.
  PUSH DX         ; leave result on stack
```

(however, as we shall see below, there will need to be some necessary boilerplate lines that conform to the particular assembler's conventions, as well as respecting the calling conventions of the high-level language we are going to use the subroutine with).

To test the assembly language program with F-PC's intrinsic assembler, we modify it slightly (to conform to the latter's notational conventions), obtaining [Listing Three].

An assembler written in Forth is simple because the mnemonics are actually `IMMEDIATE` words that execute during assembly, placing the appropriate operation codes in the parameter field of the word being defined. In the F-PC assembler, the `LOOP` mnemonics (`LOOP`, `LOOPZ`, `LOOPNZ`, etc.) expect a number on the stack, which is actually the address they loop back to (or not, depending whether an appropriate condition is satisfied). This can be supplied by an explicit label or, as in the above example, we may simply say `HERE`, which places on the stack the address of the next piece of code to be assembled; this is the very point we want to loop back to, hence `LOOP` enters the Intel opcode for `LOOP`, together with that address.

We now enter the subroutine from the keyboard and test the result.

```
CODE STIB <cr> ok
POP BX <cr> ok
POB CX <cr> POB <-WHAT?
```

Oops! A typo, do it again. Just in case, `FORGET` from `STIB` on:

```
FORGET STIB <-WHAT?

CODE STIB <cr> ok
POP BX <cr> ok
POP CX <cr> ok
XOR DX, DX <cr> ok
HERE <cr> ok
```

Listing Three

```
CODE  STIB            \ reverse bit-order
     POP BX           \ get n
     POP CX           \ get # of iterations
     XOR DX, DX       \ set n' = 0
   HERE               \ beginning of loop
     SHR BX, # 1      \ send 0'th bit of n to CF
                      \    and shift right 1 place
     RCL DX, # 1      \ shift n' left and
                      \   move CF into 0'th bit of n'
  LOOP                \ CX=CX-1, loop if CX > 0.
     PUSH DX          \ leave result on stack
NEXT  END-CODE        \ terminate definition
```

```
SHR BX, # 1 <cr> ok
RCL DX, # 1 <cr> ok
LOOP <cr> ok
PUSH DX <cr> ok
NEXT <cr> ok
END-CODE <cr> ok
```

This all looks like it entered correctly—at least the assembler did not burp. The proof of the pudding, however, is in the eating:

```
  4 7 STIB . <cr> 14 ok
 4 14 STIB . <cr> 7 ok
```

Eureka! No warts this time.

If an assembly language version of `STIB` were needed for linking with a BASIC or C program, some minor modifications would be necessary:

- the comments would have to be preceded with a semicolon `;` rather than Forth's traditional backslash `\`;
- the word `HERE` must be converted to a loop label;
- a standard header must be added, and the definition termination also changes.

[See Listing Four.]

Case conversion

Many languages contain a library function for converting a string to all upper-case letters or all lower-case ones, leaving digits and punctuation alone. The new Forth ANS standard [9] happens not to require such a routine, although most Forths contain a word analogous to `UCASE` as part of the com-

9. A copy of the final draft of the ANS Forth Standard document, *X3J14 dpANS-6* can be downloaded in several different machine-readable formats, including F-PC hypertext, Microsoft Word , or HTML, from the Web site www.taygeta.com.

Listing Four

```
Code segment word public 'CODE'  ; define the code segment
assume cs: Code
public STIB                      ; allow any routine to call it
    STIB proc near               ; reverse bit-order
         POP BX                  ; get n
         POP CX     ; get # of iterations
         XOR DX, DX ; zero n'
    HERE:    ; label beginning of loop
         SHR BX, 1  ; 0'th bit -> CF, shift right
         RCL DX, 1  ; n': shift left, CF -> 0'th bit
         LOOP HERE               ; CX=CX-1, loop if CX > 0.
         PUSH DX                 ; leave result on stack
         RET                     ; return from function call
    STIB endp                    ; terminate definition
    Code ends
         end
```

Listing Five

```
  : lcase?            ( char -- flag)  \ true if lower case
     DUP  [ CHAR]  a < ( char f1)       \ true if char < "a"
     SWAP [ CHAR]  z > ( f1 f2)         \ true if char > "z"
     OR                ( not[ flag] )   \ combine flags
     NOT ;                              \ logical not

  : UCASE             ( beg len)
      0 DO            \ work from left to right thru string
       DUP C@         ( -- adr char)   \ get character
       DUP  lcase?    ( -- adr char flag)
       32 AND         ( -- adr char  32 if lcase | 0 else)
       -              \ subtract 32 from lcase letters only
       OVER C!        \ replace modified character
   1+  LOOP           \ increment address by 1 and loop
   DROP ;             \ clean up stack
```

Listing Six

```
  : lcase? ( char -- flag)
     DUP   [ CHAR]  a  <           ( char flag1)
     SWAP  [ CHAR]  z  >           ( flag1 flag2)
     OR                                  ( not[ flag] )
     NOT                                 ( flag)
  ;
```

piling mechanism.

The first step is to choose our approach. In Microsoft Quick Basic® (QB), a string of *N* characters is stored in a contiguous sequence of *N* bytes of memory in the default data segment. It is referenced by a 4-byte string descriptor, with the first two bytes containing the length as a signed 16-bit integer, and the second two bytes the offset of the beginning of the string in the data segment. That is, Quick Basic strings can be up to 32 Kb long. Microsoft C stores strings in contiguous segments of *N*+1 bytes with the *N*+1st byte containing 0 (standard terminator), strings being referenced by the address of their first byte.

Forth, by contrast, usually deals in counted strings up to 256 bytes long, whose count is contained in the first byte. These differences between languages present a minor problem in designing subroutines that manipulate strings, since they will not work the same in Forth as in QB or C. The easiest method is to write the code in two pieces: a language-specific header and a universal body. We illustrate with headers for Forth, Quick Basic and C string-storage conventions.

What of the body code? If we write it first in high-level Forth the design becomes clear [10] [see Listing Five]. That is, we step through the string a byte at a time from beginning to end, testing whether the character is a lower-case letter or "other." If lower case, change to upper case; otherwise do nothing. The actual switch from lower to upper case is accomplished by subtracting 32d from the ASCII character code of the letter, since the upper-case letters have codes 32d smaller than their corresponding lower-case values. It is worth noting that in the words `lcase?` and `UCASE` the programming style computes the result rather than deciding it [11]. That is, while it is not always practical to avoid decisions [12], good style eschews branches wherever possible.

We test the Forth version using non-Standard but common words `$"` (save a string from the keyboard to temporary storage) and `$.` (print a counted string at address *adr* to the screen) [13]:

```
$" this + is A % lcase STring" COUNT UCASE
PAD $. THIS + IS A % LCASE STRING ok
```

The assembly language version is easy to construct. Begin with `lcase?` [Listing Six] and recode directly in assembly language [14] [see Listing Seven].

Test it. (Note: "true"—all bits set to 1— is assumed to be -1 in this example, an illustration of "environmental dependency," i.e., code that assumes two's-complement integer arithmetic.)

```
CHAR A DUP . lcase? . 65 0 ok
CHAR a DUP . lcase? . 97 -1 ok
CHAR z DUP . lcase? . 122 -1 ok
CHAR & DUP . lcase? . 38 0 ok
```

10. The ANS Forth word `WITHIN` could have been used for this test, as in
```
: lcase?      ( char -- flag=true if char is lower case)
    [CHAR] a   [CHAR] z   1+   WITHIN  ;
```
(The extra `1+` is required because ANS Forth defines `WITHIN` so that it returns `TRUE` if the limits satisfy $n_1 \le char < n_2$.) However, this would not illustrate directly how to turn primitive Forth operations into `CODE`.

11. Note that, in this example, the use of a branching construct (`IF` ... `ELSE` ... `ENDIF`) eliminates the need to store a byte if the character were upper case or a non-letter. That is, we could say
```
lcase?    IF  32 -  SWAP  C!  ELSE  DROP  ENDIF
```
However, tests reveal that most text input is predominantly lower case; hence, the time consumed in the branch dominates the unnecessary store operations.

12. J.V. Noble, *Computers in Physics*, Jul/Aug 1991, p. 386.

13. If your Forth lacks `$"` and `$.` here are their definitions:
```
: $"   [CHAR] "  WORD   PAD  OVER  C@ 1+  CMOVE  PAD ;
\ Read text up to a terminating " then move it to PAD
\(temp storage); leave the string's new address.

: $.   ( adr—)  COUNT  TYPE  ;
```

14. Several things to keep in mind about the translation: since we are working with ASCII codes, i.e., integers in the range 0..255, we can save register space by using eight-bit operations rather than 16- or 32-bit ones. (Of course, when `PUSH`ing or `POP`ping to/from the data stack, the operations appropriate to the cell size of the stack must be used.)

15. An alternate assembly definition of `lcase?` employing jumps is
```
CODE lcase?                          ( char -- f)
        POP  BX                          \ char -> BL
        XOR  AX, AX                      \ AX = false
        CMP  BL, # 97                    \ < a ?
        JL   DONE
        CMP  BL, # 122                   \ > z ?
        JG   DONE
        NOT  AX                          \ AX = true
    DONE:
        PUSH  AX                         \ flag -> TOS
    NEXT   END-CODE
```
whereas we used no jumps to perform the tests in the previous version. The jumpless version assembles to 18 bytes, whereas the one with jumps requires 18. That is, although the version with jump instructions looks much shorter than the branchless version, it is actually nearly the same length. And eliminating jumps can reduce the likelihood of having to dump the pipeline.

Listing Seven

```
CODE lcase?                  ( char -- flag)
   POP BX                   \ char -> BL
   MOV AX, BX               \ copy to AL
   SUB AL, # 96             \ AL = char - 96
   CBW                      \ sign AL -> AH = flag1
   XCHG AX, BX              \ interchange registers BH = flag1
   SUB AL, # 123            \ AL = char - 123
   CBW                      \ sign AL -> AH = flag2
   OR  AH, BH               \ AH = flag1 or flag2
   XCHG AL, AH              \ AL = ~flag
   NOT AL                   \ AL = flag
   CBW                      \ convert 8- to 16-bit flag
   PUSH AX                  \ flag -> TOS
NEXT  END-CODE              \ terminate definition
```

The preceding test went well—we can test efficiently whether a character is lower case [15]. To proceed, we next require a looping construct. The one we used in STIB will do fine, because once again the loop will execute a predetermined number of times. Again we must provide header code that places the count (string length, in bytes) in the CX register, and the address of its first byte in BX. This time, however, we identify the header code as a separate section of the assembler subroutine, in order to be able to replace it later with an appropriate equivalent that respects the conventions of a language other than Forth.

In F-PC the header will consist of the instructions:

```
POP CX              \ count in CX
POP BX              \ beginning of data in BX
PUSH DI             \ save DI (index) register
MOV DI, BX          \ start-1 in DI
```

Upon exiting, we restore DI with BX DI, as the last instruction preceding NEXT END-CODE. Conversely, a header suitable for Quick Basic would look like [16] [see Listing Eight] and the corresponding QB footer (to exit gracefully) would be

```
POP DI
POP BP       ; restore registers
```

The complete program in F-PC assembler then becomes [Listing Nine].

The subroutine is hard to read even with indented comments (which is why we prefer high-level language to assembler), but it consists of the same parts as the high-level definition: a SETUP section that gets the count and origin of data; a body that LOOPs through the string; a test that determines whether a character is a lower-case letter, and if so, modifies it to upper case; and a "footer" that restores whatever registers have been saved on the stack and exits gracefully. Note we were able to eliminate three redundant instructions:

```
XCHG AL, AH
CBW
PUSH AX
```

whose only purpose in the CODE version of lcase? was to convert an 8-bit flag to a 16-bit integer that could be left on the stack. The code for UCASE is about as terse as such a routine can be made.

Since assembler supposedly provides raw speed, it is interesting to examine timings [16]. Looking up the number of clock cycles per instruction for the Intel 80286 [17], we find [Listing Ten].

The instructions labeled "assembler directive" execute during compilation and carry no run-time overhead. Since the header and footer are executed once, their 25 clock cycles are immaterial for reasonably long input strings. Converting a lower-case to an uppercase letter evidently requires 42 clock cycles, i.e., about 1.3 µsec on a 33 MHz machine. The test loop

```
: TEST0
  0 DO  PAD UCASE LOOP ;
: TEST1
  0 DO  10000 TEST0 LOOP ;
```

allows us to iterate enough times to get meaningful data: saying 10 TEST1 iterates 105 times. The time to convert 45 characters is 7 seconds, giving a per-character time of 1.5 µsec, in reasonable agreement with the estimate from machine cycles. This is 24 times faster than the high-level Forth version; optimization is definitely worthwhile when we have many strings to convert.

For completeness, here is a version that works with (zero-terminated)

16. *Note:* if we were trying to generate the same function for linking to C, we would have to take into account the zero-terminated structure of strings in C, probably using a different looping method, since the count would not be readily available.
16. Abrash, op. cit., discusses in detail the pitfalls of assuming the instruction timings given by Intel.
17. Actually the test was performed on a 3086SX-33 machine.

Listing Eight

```
PUSH BP                         ; save BP
MOV BP, SP                ; use BP as a stack pointer
PUSH DI                         ; save DI register
MOV BX, 6 [ BP]           ; address of string descriptor to BX reg
                          ; Note: don't need to initialize CX
MOV CX, 0 [ BX]           ; count in CX reg
ADD BX, 2                 ; offset to string origin in BX
```

Listing Nine

```
CODE UCASE                  \ start header
   POP CX                        \ get count
   POP BX                        \ get origin
   PUSH DI                       \ save DI
   MOV DI, BX               \ end header, start body
HERE                             \ begin loop
   INC DI                             \ point to next byte
   MOV BL, 0 [ DI]                    \ get byte
      MOV AX, # 96                       \ test case
      SUB AL, BL
      CBW
      XCHG AX, BX
      SUB AL, # 123
      CBW
      AND AH, BH                          \ AH = FF|0
   AND AH, # 32                       \ AH = 32 if lcase, 0 else
   SUB 0 [ DI], AH                    \ convert letter in $
   LOOP                          \ loop if CX > 0
                            \ end body
                            \ footer
   POP DI                           \ restore DI
   NEXT                     \ end footer
END-CODE
```

strings in C. There are two obvious ways to approach the problem: first, modify the loop in UCASE so it terminates when the byte fetched is 0 (not ASCII 0). Alternatively, if we had a fast way to determine the string's length, we could use the preceding code unmodified. Now, we know only the beginning address of a C string, so to determine its length we must search it until we find the terminating character, incrementing a counter as we go. In high-level Forth, the subroutine is [Listing Eleven] and is very slow. Given a function to compute the length of a zero-terminated string, the revised upper-case function is virtually identical to its predecessor [Listing Twelve].

To test these, we need words that input and print C-like strings [Listing Thirteen].

A quick interactive test [is shown in Listing Fourteen].

Of course, if we had to replicate the steps and tests of GET_LEN in assembler, it would obviously be better to rewrite UCASE.C entirely. Fortunately, the 80x86 chips have a special instruction pair, SCASB and REPNZ, that speed up certain

Listing Ten

```
CODE UCASE          \ 0 (assembler directive)
   POP CX                  \ 5
   POP BX                  \ 5
   PUSH DI                 \ 3
   MOV DI, BX              \ 2
                    \ total = 15 for header
HERE                \ 0 (assembler directive)
   INC DI                  \ 2
   MOV BL, 0 [ DI]         \ 5
   MOV AX, # 96            \ 2
   SUB AL, BL              \ 2
   CBW                     \ 2
   XCHG AX, BX             \ 3
   SUB AL, # 123           \ 3
   CBW                     \ 2
   AND AH, BH              \ 2
   AND AH, # 32            \ 3
   SUB 0 [ DI] , AH        \ 7
   LOOP                    \ 9
                     \ total = 42 for body
   POP DI                  \ 5
   NEXT                    \ 5 (depends on the Forth)
                     \ total = 10 for footer
END-CODE             \ 0 (assembler directive)
```

Listing Eleven

```
: GET_LEN               ( beg -- len)
       DUP              ( beg beg)
       BEGIN                  \ start indefinite loop
            DUP C@      \ get char
            0<>         ( beg adr flag)
       WHILE   1+       ( beg adr+1)
       REPEAT           (beg end+1)
                        \ loop until character is 0
       SWAP  -          ( -- len)    \ compute length
;
```

Listing Twelve

```
: UCASE.C          ( beg --)
       DUP   GET_LEN
       0 DO             \ work left to right thru string
       DUP C@           ( -- adr char)      \ get char
       DUP  lcase?      ( -- adr char flag)
     32 AND             ( -- adr char  32 if lcase | 0 else)
     -             \ subtract 32 from lcase letters only
     OVER C!       \ replace modified character
 1+  LOOP          \ increment address by 1 and loop
 DROP ;            \ clean up stack
```

string operations. The Forth assembler definition using these instructions would then be [Figure Fifteen].

This is a bit long and complicated, and no doubt will get longer and more complex when the boilerplate headers and footers that respect C conventions are added. Unless there is a specific need for a function that determines the lengths of zero-terminated strings (and, for all I know, one may be available), there does not seem to be any reason to factor out this functionality, merely to re-use the code designed for counted strings. Here is a situation where recoding `UCASE.C` from scratch is the more efficient approach.

Once again we begin by prototyping in high-level Forth,

Listing Thirteen

```
: $0"   ( -- adr)         \ input 0-terminated string
       [CHAR] "  WORD     \ get input
       DUP   ( $adr $adr)
       1+  PAD  ROT  C@   \ get length
       DUP >R             \ save it temporarily
       CMOVE              \ move text to scratchpad
       PAD  R>  OVER +    ( -- beg end+1)
       0 SWAP  C!         \ terminate with 0
;

: $0.   ( adr --)         \ print 0-terminated string
       DUP  GET_LEN  TYPE  ;
```

Listing Fourteen

```
   $0" Here is a test string of 37 characters!"  ok
   GET_LEN . 37  ok
   PAD $0.  Here is a test string of 37 characters!  ok
```

Listing Fifteen

```
HEX
CODE    GET_LEN     ( adr -- len) \ get length of 0-terminated $
        POP BX                    \ adr -> BX
        PUSH DI           \ save state
        PUSH ES           \ save "extra" segment descriptor
        MOV AX, DS        \ there is no MOV ES, DS instruction
        MOV ES, AX        \ point to data segment
        MOV CX, # FFFF    \ largest possible string
        MOV DI, BX        \ load offset
        XOR AL, AL        \ AL = 0
        REPNZ  SCASB      \ go thru $ until 0 byte found
        SUB BX, DI        \ compute length
        NEG BX
        POP ES                    \ restore state
        POP DI
        PUSH BX           \ result on stack
NEXT END-CODE
DECIMAL
```

Listing Sixteen

```
: UCASE.C          ( beg --)
       BEGIN                          \ start indefinite loop
            DUP C@                    ( -- adr char)
            DUP 0<>                   ( -- adr char flag)
       WHILE                          \ haven't reached end
            lcase?                    ( -- adr flag)
            -32  AND                  ( -- adr   -32 | 0)
            OVER  +C!                 \ modify char in place
            1+                        ( -- adr+1)
       REPEAT                         \ loop until char = 0
       DROP                           \ clean up stack
;
```

then translate to CODE. We want to hybridize GET_LEN and UCASE.C from before, i.e., replace the definite loop with an indefinite one. [Listing Sixteen]

This is easily tested using our zero-terminated string tools [Listing Seventeen].

The assembler version is easily coded. The use of CBW ("convert byte to word") avoids decisions by computing a flag (in the upper half of the AX register) based on the sign of the subtraction operation. [Listing Eighteen]

Micro-mini assembler

Although I have discussed the use of the Forth assembler in the context of rapid machine code development and/or as a propaganda device to interest outsiders in Forth, of course one should not forget that it is a useful tool in the Forth programmer's arsenal. In my own work, I have not worried too much about the fact that Forths tend to run somewhat slower than optimized C programs, because I know that if I really need to step on the gas by hand coding an inner loop, it will not take much extra effort. (There was a time, not so many years ago, when I got so carried away with that approach that I would define words in CODE at the drop of a hat, just because it was so easy. Needless to say, my work was cut out for me later on when I had to port the programs to ANS-compatible Forths. One mustn't lose one's head by over CODEing.)

When memory is limited and only a few CODE words need to be defined, rather than load the entire assembler it pays to insert the opcodes directly into the body of the code word. These are usually byte-sized numbers in hexadecimal format, and can be inserted with C, as in (suitable for F-PC)

```
CODE  MY@   HEX
  5B C,  FF C,  77 C,
NEXT  END-CODE
```

Listing Seventeen

```
$0" This is a fairly long test string of 59 characters' length."  ok
get_len . 59  ok
pad $0. This is a fairly long test string of 59 characters' length. ok
pad ucase.c  ok
pad $0. THIS IS A FAIRLY LONG TEST STRING OF 59 CHARACTERS' LENGTH. ok
```

Listing Eighteen

```
  CODE UCASE.C
     MOV DX, DI     \ save DI (in DX)
     POP DI         \ DI = beg
  1 $:              \ label to return to
     MOV BL, 0 [ DI] \ get byte
     CMP BL, # 0    \ is it 0 ?
     JZ  2 $        \ jump to end if 0
     MOV AX, # 96   \ 97d is ASCII 'a '
     SUB AL, BL     \ is the byte < 'a' ?
     CBW                 \ if BL >= 97 then AH = FFh, else AH = 0
     XCHG AX, BX
     SUB AL, # 123  \ is the byte > 'z' ?
     CBW                  \ if AL <= 122 AH = FFh; else AH = 0
     AND AH, BH     \ AH = FFh if 'a' <= byte <= 'z', else AH = 0
     AND AH, # 32   \ AH = 32 or 0
     SUB 0 [ DI] , AH \ convert byte in string
     JMP  1 $       \ loop
  2 $:              \ end
     MOV DI, DX     \ restore DI
     NEXT
  END-CODE
```

Listing Nineteen

```
\ Micro-mini assembler suitable for F-PC
HEX
: <%      HEX                       \ base 16
          BEGIN    BL  WORD  %NUMBER
          WHILE    DROP  C,
          REPEAT   2DROP  DECIMAL   \ restore base
          HERE  1+  @  3E25  <>  ABORT" Missing %> !"
;  IMMEDIATE
DECIMAL
\ Note: <% xx xx xx xx %> in a CODE definition assembles those (hex) bytes
\ Usage:  CODE  MY@  <%  5B FF 37 %>  NEXT  END-CODE
\
\ Note: to make the above work in ANS Forth we need to define %NUMBER in terms
\       of >NUMBER.
\       : %NUMBER     0.0 ROT  COUNT  >NUMBER  NIP   ;
```

Listing Twenty

```
    DECLARE SUB sphbes ()
    DEFDBL S-X
    DIM SHARED xj(40), x
    INPUT "What is x"; x
    CALL sphbes
    FOR n% = 0 TO 9
        PRINT "j"; LTRIM$(STR$(n%)); "  ("; x; ")", xj(n%)
    NEXT
    END

DEFDBL S-X
    SUB sphbes
    xj(40) = 0!
    xj(39) = 1!
    sum = 2 * 39 + 1
    FOR n% = 39 TO 1 STEP -1
        temp = xj(n%) * (2 * n% + 1) / x - xj(n% + 1)
        xj(n% - 1) = temp
        sum = sum + (2 * n% - 1) * temp * temp
    NEXT
    xnorm = 1 / SQR(sum)
    FOR n% = 0 TO 9
        xj(n%) = xj(n%) * xnorm
    NEXT
    END SUB
```

Listing Twenty-one

```
\ data structures
10  REAL*8  #CELLS  1ARRAY  JBES{      \ holds j0-j9

FVARIABLE  SUM                         \ temps to off-load from fp stack
FVARIABLE  X

: SETUP    ( F: x -- 0 1 )  ( -- 79)
     X DF!   79 S>F  SUM DF!
     F0.0 F1.0    79  ;

: NORMALIZE     SUM  DF@  FSQRT  1/F
      10 0   DO   FDUP   JBES{  I }  DUP  DF@  F*   DF!   LOOP
      FDROP  ;

: DO_X=0    FDROP  F1.0  JBES{  0 } DF!
            10 1  DO    F0.0   JBES{  I }   DF!    LOOP   ;

: ITERATE   ( F:  jn+1 jn -- jn jn-1)  ( 2n+1 -- 2n-1)
      DUP  S>F   FOVER  F*       ( F:  jn+1 jn jn*[ 2n+1] )
      X DF@  F/    FROT  F-      ( F:  -- jn jn-1)
      FDUP  F^2                  ( F:  -- jn jn-1 jn-1^2 )
      2-  DUP                    ( -- 2n-1 2n-1 )
      S>F  F*
      SUM DF@   F+   SUM DF!   ;

: SPHBES  ( F: x --)
      FDUP   F0=
      IF     DO_X=0   EXIT    THEN
      SETUP
      11 39 DO   ITERATE   -1 +LOOP
      0  9  DO   ITERATE
                 FDUP   JBES{  I }  DF!
      -1 +LOOP
      DROP   FDROP  FDROP        \ clean up stacks
      NORMALIZE  ;
```

If there are more than a few such words, but one would prefer not to load the assembler, the following word may be of use. [Listing Nineteen]

Spherical Bessel functions

Here is an example of a fairly complex subroutine from a number-crunching application. It was necessary to code this function in assembler because it was used many times.

If one only needs a single spherical Bessel function, $j_n(x)$, it is usually best just to compute it in terms of $\sin(x)$, $\cos(x)$ and polynomials in $1/x$. However, when more than one is needed, especially functions of high order, the most practical approach is recursion. The obvious method of upward recursion, based on the relation

$$j_{n-1}(x) = (2n+1)x^{-1}j_n(x) - j_{n+1}(x)$$

and starting with explicit formulae for $j_0(x)$ and $j_1(x)$, is unstable and rapidly loses numerical precision. We therefore employ the downward recursion recommended by Abramowitz and Stegun [18], with starting values (for some large N)

$$j_N = 1, \quad j_{N+1} = 0$$

then apply the relation

$$\sum_{k=0}^{N} (2k+1)\left[jk(x)\right]^2 = 1.$$

to obtain the normalization. In QuickBasic the program might look like [Listing Twenty], whereas a Forth version is [Listing Twenty-one].

Translating this routine to assembler will be the *pièce de resistance* of this article. It is rather long, and represents the upper limit of what is reasonable to hand code as a single subroutine in the never-ending search for speed. We shall maintain temporary values and intermediate expressions on the intrinsic stack of the floating-point co-processor to minimize transfers to/from the (slower) main memory. The public-domain Forth F-PC does not come with 80x87 extensions to its assembler. Therefore, to assemble and test the subroutine, we must choose one of the following courses:

- add the necessary extensions to the F-PC assembler (Robert L. Smith has done this in creating the floating-point extensions ffloat.seq available on various Forth archives);
- use the Micro-mini assembler described above;
- employ a Forth with a more complete assembler, such as Tom Zimmer's Win32Forth.

The floating-point units associated with Intel microprocessors possess an intrinsic eight-deep stack [19]. Upon entering the subroutine, the on-chip stack must be initialized to contain nothing, which we visualize as

```
st(7)      ...
st(6)      ...
st(5)      ...
st(4)      ...
st(3)      ...
st(2)      ...
st(1)      ...
st(0)      ...
```

The first steps are initialization, following which the FPU stack will contain x, the argument of the Bessel function(s), as well as the initial values of j_n, j_{n+1}, and whatever else may be needed. In fact, it looks like

```
st(7)      ...
st(6)      ...
st(5)      ...
st(4)      x
st(3)      sum
```

18. M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions* (Dover Publications, Inc., New York, 1965) p. 452.

19. The stack notation (87: --) refers to the eight-deep FPU-intrinsic stack (the Intel FPU began as a separate chip with the designation 8087/80287/80387 before being combined onto the 80486 and Pentium® series).

Listing Twenty-two

```
st(7) ...                  st(7)     ...
st(6)     ...                     st(6)     ...
st(5)     ...                     st(5)     ...
st(4)     x                st(4)     x
st(3)     sum                     st(3)     sum + (2n+1)*jn*jn
st(2)     2n+1                    st(2)     2n-1
st(1)     jn+1                    st(1)     jn
st(0)     jn               st(0)     jn-1
```

Listing Twenty-three

```
finit                                   \ clear fpu stack
mov     ecx, FSP [ edi]                 \ get fstack ptr
sub     ecx, # B/FLOAT                  \ decrement by data size
js      L$2                             \ -> error handler
fld     FSIZE FSTACK [ ecx] [ edi]  ( 87: x)
mov     FSP [ edi] , ecx                \ adjust fstack ptr
push    ebx                             \ TOS -> mem
push    # 4F                        \ 79d=4Fh on data stack
fldz                                            \ fload 0     ( 87: 0 x)
fild    dword  0 [ esp]                 \ 79d -> st(0)
fldz                                            \ fload 0
fld1                                            \ fld 1 ( 87: 1 0 79 0 x)
pop     ebx                             \ ebx = 79
( 87: jn jn+1 2n+1 sum x)
```

Listing Twenty-four

```
       FXCH ST(1)    FLD  ST(1)    FMUL  ST(0), ST(3)

st(7)  ...                  ...                 ...
st(6)  ...                  ...                 ...
st(5)  ...                  x                   x
st(4)  x                    sum                 sum
st(3)  sum                  2n+1                2n+1
st(2)  2n+1                 jn                  jn
st(1)  jn                   jn+1                jn+1
st(0)  jn+1                 jn                  jn*(2n+1)


       FLD ST(0)            FMUL ST(0), ST(3)   FADDP ST(5), ST(0)

st(7)  ...                  ...                         ...
st(6)  x                    x                           ...
st(5)  sum                  sum                         x
st(4)  2n+1                 2n+1                        sum'
st(3)  jn                   jn                          2n+1
st(2)  jn+1                 jn+1                        jn
st(1)  jn*(2n+1)            jn*(2n+1)                   jn+1
st(0)  jn*(2n+1)            jn*(2n+1)*jn        jn*(2n+1)


       FDIV ST(0), ST(5)    FSUBRP ST(1), ST(0)        FLD1

st(7)  ...                          ...                                 ...
st(6)  ...                          ...                                 ...
st(5)  x                            ...                                 x
st(4)  sum'                         x                                   sum'
st(3)  2n+1                         sum'                                2n+1
st(2)  jn                           2n+1                                jn
st(1)  jn+1                         jn                                  jn-1
st(0)  jn*(2n+1)/x          jn-1                              1


       FSUB ST(3), ST(0)    FSUBP  ST(3), ST(0)

st(7)  ...                          ...
st(6)  ...                          ...
st(5)  x                            ...
st(4)  sum'                         x
st(3)  2n                           sum'
st(2)  jn                           2n-1
st(1)  jn-1                         jn
st(0)  1                            jn-1
```

Listing Twenty-five

```
   fxch    st(1)           ( 87: jn+1 jn k=2n+1 sum x)
   fld     st(1)           ( 87: jn jn+1 jn k sum x)
   fmul    st(0), st(3)    ( 87: k*jn jn+1 jn k sum x)
   fld     st(0)           ( 87: k*jn k*jn jn+1 jn k sum x)
   fmul    st(0), st(3)    ( 87: k*jn^2 k*jn jn+1 jn k sum x)
   faddp   st(5), st(0)    ( 87: k*jn jn+1 jn k sum' x)
   fsubpr  st(1), st(0)    \ this is a sp. error in 486asm.f
\ ^^^^^^                   \ -- should be fsubrp
   fld1
   fsub    st(3), st(0)
   fsubp   st(3), st(0)    ( 87: jn-1 jn 2n-1 sum' x)
```

Listing Twenty-six

```
finit  ok
3.7e0  ok
ITERATE  ok.
. 79  ok
fpush f. 1.00000  ok
fpush f. .000000  ok
fpush f. 79.0000  ok
fpush f. 79.0000  ok
fpush f. 3.70000  ok
see iterate
ITERATE IS CODE
         3712C 8B8F3CC70100     mov     ecx , FSP [ edi]
         37132 83E908           sub     ecx , # 8
         37135 0F881F000000     js      ITERATE
         3713B DD84394CC70100   fld     double FSTACK [ ecx] [ edi]
         37142 898F3CC70100     mov     FSP [ edi] , ecx
         37148 53               push    ebx
         37149 6A4F             push    , # 4F
         3714B DB0424           fild    dword [ esp]
         3714E D9C0             fld     ST(0)
         37150 D9EE             fldz
         37152 D9E8             fld1
         37154 5B               pop     ebx
         37155 E908000000       jmp     ITERATE
         3715A C7C650CF0100     mov     esi , # 1CF50
         37160 03F7             add     esi , edi
```

Listing Twenty-seven

```
3.7e0  ok
iterate  ok.
.s [ 1]  77  ok.                    \ left 2n-1 on data stack to check it
fpush f. 21.3514  ok.
fpush f. 1.00000  ok.
fpush f. 77.0000  ok.
fpush f. 158.000  ok.       \ This is the sum. Note it should be 79.
fpush f. 3.70000  ok.
see iterate
ITERATE IS CODE
         3712C 9BDBE3           finit
         3712F 8B8F3CC70100     mov     ecx , FSP [ edi]
         37135 83E908           sub     ecx , # 8
         37138 0F8840000000     js      ITERATE
         3713E DD84394CC70100   fld     double FSTACK [ ecx] [ edi]
         37145 898F3CC70100     mov     FSP [ edi] , ecx
         3714B 53               push    ebx
         3714C 6A4F             push    , # 4F
         3714E DB0424           fild    dword [ esp]
         37151 D9C0             fld     ST(0)    \ wrong step
         37153 D9EE             fldz
         37155 D9E8             fld1
         37157 5B               pop     ebx
         37158 FFCB             dec     ebx
         3715A DD949FDC6F0300   fst     double JBES{ [ edi] [ ebx* 4]
         37161 D9C9             fxch    ST(1)
         37163 D9C1             fld     ST(1)
         37165 D8CB             fmul    ST , ST(3)
         37167 D9C0             fld     ST(0)
         37169 D8CB             fmul    ST , ST(3)
         3716B DEC5             faddp   ST(5)
         3716D D8F5             fdiv    ST , ST(5)
         3716F DEE1             fsubrp  ST(1)
         37171 D9E8             fld1
         37173 DCEB             fsub    ST(3) , ST
         37175 DEEB             fsubp   ST(3)
         37177 FFCB             dec     ebx
         37179 E908000000       jmp     ITERATE
         3717E C7C650CF0100     mov     esi , # 1CF50
         37184 03F7             add     esi , edi
```

```
st(2)      2n+1
st(1)      jn+1
st(0)      jn
```

At each subsequent iteration the stack transforms as [Listing Twenty-two].

Let us begin with the initialization steps [Listing Twenty-three]. The initialization clears the FPU stack and moves x from the in-memory fstack to the fpu. (This part is taken directly from `float.f`'s word fpop.) Finally, numeric constants are loaded.

Next we consider what happens during each iteration: we must pay careful attention to the FPU stack because there are five items on it after initialization. We note we shall need the factor $(2n+1)j_n$ in two places: first, to calculate j_n; and second, to calculate the next term in the sum. To work out the steps, we show the FPU stack after each machine instruction [Listing Twenty-four].

That is, the complete sequence of instructions that performs one iteration is [Listing Twenty-five].

Now, how can we test this to be sure it is correct? The beauty of testing an assembly language subroutine within the Forth environment is that no linking step is required. Thus we can assemble larger and larger subsets of the `CODE` word, testing each portion and `FORGET`ting it to test the next iteration. (Assuming, that is, we have not caused the system to crash in one of the experiments!) Here are some actual test runs performed while getting the word `ITERATE` off the launching pad [Listing Twenty-six].

What I have done here is show the steps needed to hand-initialize the system before invoking the word (`ITERATE`) being tested. I then said `ITERATE` to run the part that had been assembled so far, and examined the FPU stack with the `float.f` words `fpush f.` to be sure all had gone as expected. The final step was to disassemble `ITERATE` to record what it consisted of so far.

The eventual idea is to simulate a `BEGIN` … `UNTIL` loop in assembler. Many Forth assemblers provide macros for this purpose, but since my aim was to create a subroutine that could be ported easily to another high-level language (given the proper boilerplate header and footer), I did not wish to avail myself of Forth-specific macro facilities.

The next test stage consisted of adding the machine code to perform one iteration. The result is shown below. I have not yet installed either the looping construct or the code to clean up the stacks afterward. [Listing Twenty-seven]

Note that at this point I added the `finit` step into the subroutine instead of performing it by hand. But this time all did not go well. The sum that has been computed is wrong—it is 158 rather than 79 as it should have been. The reason is easy to see: I had a misconception that this term would be computed last, using $2n$-1 and j_{n-1} rather than before j_{n-1} using $2n$+1 and j_n. The fix is simple, namely to modify the initialization sequence to load 0 rather than $2n$+1 for the initial *sum*. Here is the fixed result [Listing Twenty-eight].

Now all is correct. We are ready to add the code to simulate `BEGIN ... UNTIL` as well as that needed to clean up the various stacks. Note that, at the beginning of an iteration, the current value of the Bessel function (not yet properly normalized, of course) gets stored in its proper array element of the array `jbes{` . This is done by computing the base address using the phrase `jbes{ 0 }` [20] which is then added to the offsets indexed by registers ebx and edi. Note that the array index seems to be multiplied by 4 (bytes) as for 32-bit precision. However, at this storage step, the value in ebx is $2n$ because ebx has been decremented once. So in fact the subroutine is written to store 64-bit floating-point numbers—vital because the magnitude of the un-normalized functions (not to mention that of the normalization *sum*) can grow easily past the numbers accommodated in IEEE 32-bit precision.

In fact, the first `dec ebx` instruction (leaving $2n$ in `ebx`) marks the beginning of the loop. The second `dec ebx` instruction marks the last computational step of the loop. We label the beginning of the loop with the assembler's local label facility (the phrase `L$1:`) and use the Intel `jns` ("jump not sign") instruction to loop back to it when the decrement operation has not changed the algebraic sign of the index in the ebx register (that is, while $2n$-1 > 0).

Finally we must clean up the stacks. The exit value of the index (-1) needs to be replaced in the `ebx` register (which is used as the top of the data stack by Win32Forth) by whatever was on top of the stack before entering the subroutine. This is accomplished by the `pop ebx` instruction. Since it does not particularly matter when this is done, we perform this last.

20. This notation was introduced in my book *Scientific Forth: a modern language for scientific computing* (Mechum Banks Publishing, Ivy, VA 1992) and has been adopted as standard for the Forth Scientific Subroutine Library Project organized by Skip Carter.

Listing Twenty-eight

```
3.7e0  ok
iterate  ok.
.s [ 1]  77  ok.
fpush f. 21.3514  ok.
fpush f. 1.00000  ok.
fpush f. 77.0000  ok.
fpush f. 79.0000  ok.
fpush f. 3.70000  ok.
see iterate
ITERATE IS CODE
         3712C 9BDBE3           finit
         3712F 8B8F3CC70100     mov     ecx , FSP [ edi]
         37135 83E908           sub     ecx , # 8
         37138 0F8840000000     js      ITERATE
         3713E DD84394CC70100   fld     double FSTACK [ ecx] [ edi]
         37145 898F3CC70100     mov     FSP [ edi] , ecx
         3714B 53               push    ebx
         3714C 6A4F             push    , # 4F
         3714E D9EE             fldz          \ repaired step
         37150 DB0424           fild    dword [ esp]
         37153 D9EE             fldz
         37155 D9E8             fld1
         37157 5B               pop     ebx
         37158 FFCB             dec     ebx
          3715A DD949FDC6F0300   fst     double JBES{ [ edi] [ ebx* 4]
         37161 D9C9             fxch    ST(1)
         37163 D9C1             fld     ST(1)
         37165 D8CB             fmul    ST , ST(3)
         37167 D9C0             fld     ST(0)
         37169 D8CB             fmul    ST , ST(3)
         3716B DEC5             faddp   ST(5)
         3716D D8F5             fdiv    ST , ST(5)
         3716F DEE1             fsubrp  ST(1)
         37171 D9E8             fld1
         37173 DCEB             fsub    ST(3) , ST
         37175 DEEB             fsubp   ST(3)
         37177 FFCB             dec     ebx
         37179 E908000000       jmp     ITERATE
         3717E C7C650CF0100     mov     esi , # 1CF50
         37184 03F7             add     esi , edi
```

The only number we wish to retain from the FPU stack is the sum, so we simply pop the top three items with three repetitions of the instruction `fstp st(0)`; then we move the *sum* to the in-memory fstack (simply copying the code sequence from `fpush` for this purpose); and finally we drop *x* from the FPU stack with one more repetition of `fstp st(0)`.

Believe it or not, when I added this code and tested the high-level word `sphbes` given in the listing below, it worked perfectly first crack out of the box. The entire test sequence, including the mistake I had to correct, lasted 15–20 minutes. I do not believe MASM® or TASM® could come within an order of magnitude of this time.

With the completion of the spherical Bessel function routine, I end this call to assembly. Class dismissed.

Appendix

```
FALSE [ IF]

Regular spherical Bessel functions j_n(x), n=0-39

(Assembly language version suitable for Win32Forth)

© J.V. Noble 1999. May be used for any purpose as long
  as this copyright notice is maintained.

Uses Miller's method of downward recursion, as described
in Abramowitz & Stegun, "Handbook of Mathematical Functions"
10.5 ff. The recursion is

     j(n-1) = (2n+1) j(n) / x  - j(n+1)

The downward recursion is started with j40 = 0, j39 = 1 . The
resulting functions are normalized using

     Sum (n=0 to inf) {  (2n+1) * jn(x)^2 }  = 1 .

Usage:
       To calculate j0-j39 say, e.g.,

              3.0e0  sphbes

       To access/display a value say, e.g.,

              jbes{  3 }   F@   F.   .1520516620  ok
[ THEN]

marker  -jbes
include arrays.f

40 long  1 dfloats  1array  jbes{
FVARIABLE x

HEX
code ITERATE    ( f: x -- )
  \ initialization
       finit                                  \ clear fpu stack
       mov     ecx, FSP [ edi]
       sub     ecx, # B/FLOAT
       js      L$2                            \ -> error handler
       fld     FSIZE FSTACK [ ecx] [ edi]  ( 87: x)
       mov     FSP [ edi], ecx
       push    ebx
       push    # 4F                           \ 79d on data stack
       fldz                                          ( 87: 0 x)
       fild    dword  0 [ esp]             ( 87: 79 0 x)
       fldz
       fld1                                ( 87: 1 0 79 0 x)
```

```
      pop     ebx                          \ ebx = 79
      ( 87: jn jn+1 2n+1 sum x)   \ end of initialization

L$1:  dec     ebx                  \ loop begins here
      fst     double jbes{  0 } [ ebx*4] [ edi]
\     fwait                                \ may be needed
      fxch    st(1)            ( 87: jn+1 jn k=2n+1 sum x)
      fld     st(1)            ( 87: jn jn+1 jn k sum x)
      fmul    st(0), st(3)     ( 87: k*jn jn+1 jn k sum x)
      fld     st(0)            ( 87: k*jn k*jn jn+1 jn k sum x)
      fmul    st(0), st(3)     ( 87: k*jn^2 k*jn jn+1 jn k sum x)
      faddp   st(5), st(0)     ( 87: k*jn jn+1 jn k sum' x)
      fdiv    st(0), st(5)     ( 87: k*jn/x jn+1 jn k sum' x)
      fsubpr  st(1), st(0)    \ this is a sp. error in 486asm.f
   \ ^^^^^^                   \ -- should be fsubrp
      fld1
      fsub    st(3), st(0)
      fsubp   st(3), st(0)     ( 87: jn-1 jn 2n-1 sum' x)
      dec     ebx
      jns     L$1                  \ loop ends here
                                      ( 87: j0 j1 -1 sum x)
      fstp    st(0)            ( 87: j1 1 sum x)
      fstp    st(0)            ( 87: 1 sum x)
      fstp    st(0)            ( 87: sum x)
      mov     ecx, FSP [ edi]                  \ sum->fstack
      fstp    FSIZE FSTACK [ ecx] [ edi]
      fwait
      add     ecx, # B/FLOAT
      mov     FSP [ edi], ecx
      fstp    st(0)            ( 87: x -- )
      pop     ebx              ( -1 --)
      jmp     L$3

L$2:  mov     esi, # ' FSTKUFLO >body  \ error handler
      add     esi, edi

L$3:  next,
      end-code

DECIMAL

: DO_X=0     \ handle the special case x=0
      FDROP  F1.0  JBES{  0 }  DF!
      10 1  DO    F0.0   JBES{  I }    DF!    LOOP   ;

: NORMALIZE     ( f: sum --)
    FSQRT  F1.0  FSWAP  F/
    39 0   DO   FDUP   JBES{  I }   DUP  F@  F*   F!   LOOP
    FDROP  ;

: SPHBES  ( f: x --)
    FDUP   F0=             \ x=0 ?
    IF     DO_X=0   ELSE    ITERATE  NORMALIZE   THEN  ;
```

Virtual Hardware Definition Language

Using Forth as a VHDL

Abstract

A set of VHDL extensions to Forth lets programmers define hardware in the same language with which they write software. Hardware defined in Forth can be verified by executing the hardware-definition words at the command line or by writing special Forth words to test their operation. The use of the same language for hardware and software simplifies the task of swapping hardware and software functions during optimization.

Introduction

Computer-aided design has become an essential part of product development, and several different hardware definition languages (HDLs) are marketed for that purpose, but I wanted to define the hardware in the same language the software was written in. We have been using Forth to define PAL equations for about ten years using a set of extensions to Forth called CARMAP. When we started using complex programmable logic devices (CPLDs), it seemed more logical to extend CARMAP than to buy an off-the-shelf compiler and learn a new language.

With each improvement, CARMAP has moved closer to being a complete high-level design system. A program to fit the design into the PLD was added along with a method of defining what inputs an output needs. When the upgrades were finished, the compiler could automatically reduce a virtual description to logical pieces, and fit them into the macrocells of the PLD.

After the fitting is complete, the macrocells and their outputs have to be placed and arranged so all outputs can be routed to the places they are required. This problem is somewhat like solving a multi-dimensional Rubik's Cube.

Why use Forth to simulate hardware?

1. A software model can be completed much faster than hardware.
2 Application code can be tested before the hardware is designed.
3. It is easy to display or modify internal states.
4. Diagnostic macros can be easily implemented.
5. Resources can be optimized early in the design.

Why use Forth as a VHDL?

1. To reduce the time needed to create the system.
2. So Forth can be the hardware description language.
3. So the project can use one uniform language.
4. To support the extensibility of the design.
5. To enable interactive hardware design.

Designing logic with the Forth VHDL

1. Write a software simulation of the design.
2. Test the design.
3. Convert the software simulation into a hardware definition.
4. Compile the hardware definition into logic equations.
5. Fit the logic equations into the device.
6. Verify that the logic equations work correctly.
7. Route the signals and assign the I/O pins.
8. Convert the routed design into a fuse map.

Simulation

The simulation of a design allows interactive analysis of many aspects of the hardware including complexity, functionality, timing, and performance. If the application program for the proposed hardware is also written in the same language as the hardware, hard and soft components can be interchanged during optimization.

The software model

The software model is like a black box, it doesn't matter how it works as long as it works correctly. The main advantage of the software model is that structural details of the PLD can be ignored as ideas are evaluated early in the design stage.

The hardware definition

After the software model has been evaluated, the design is turned into a hardware definition. The hardware definition is an expanded version of the software model. Programs will run a little slower on it, but they should function the same.

For a design to match up with the device structure, it must be partitioned correctly. Partitioning is an intuitive process that is difficult to automate, so information relating to hardware structure needs to be included in the definition.

The conversion to the hardware definition involves breaking complex functions into smaller parts that will fit in a single layer of logic. A library of words to expand complex functions could be built to aid in this task. Global variables must be created for the output nodes of all the logic blocks, and procedures must be written that will function the same as the components would behave.

The inputs and outputs of the procedures are passed via the global variables that hold the state of the model. These variables have three parts: the first holds the present state, the second holds the future state, and the third holds the don't-care flags. The global variables also contain information about register clocking and propagation delays.

John R. Hart • Tempe. Arizona
jhart@testra.com • www.testra.com

The author is doing design work for Testra Corp., which is manufacturing an integrated motion control system for industrial and robotic applications. The system is based on a Forth processor designed using the HDL described here.

Verification of the model

Debugging the simulation of a design will require a set of tools that are "tuned" to the characteristics of the design. In a non-extensible language, this might be done using some form of macros; but when debugging a simulation in Forth, a lot of the tools exist even before the job is started. Simple things can be interactively tested by keying in and running short programs.

The simulation process involves executing all the simulation procedures, then copying all the future states into the present state. The relationship of timing and propagation is established by the order in which the state of the global variables is changed.

Things to consider when creating the hardware model

1. *Truth table size*. The size of a truth table is 2^n where *n* is the number of inputs. A function with more than 20 inputs will take a long time to compile, and it should be factored into smaller parts to reduce the number of inputs.
2. *Input relationships*. The inputs needed for an output can be specified to reduce the initial truth table size.
3. *Specify don't-care terms*. In many cases, there are places in a function table that are not used. If the unused space is flagged as don't-care, a simpler solution with a reduced number of terms may be possible.

How the logic compiler works (CARMAP)

The logic compiler converts each of the functions described in Forth into a set of logic equations for each output bit of the function. This is a conceptually simple process that involves expanding the function into a truth table and then reducing the number of terms in the truth table to the minimum.

The function is mapped into the truth table using the inputs that are related to the output. After the function has been mapped, the table is scanned for unused inputs. If any unused inputs are found, they are removed from the table. Each input that is removed cuts the table size by half.

The truth table is then converted into logic equations by an exhaustive scanning process that tries all possible combinations of inputs and compares them with the truth table. The first step is to search the table for a true output. When an output is found, all sets in which it resides are tested for correlation with the other outputs. The largest true set is saved, and the bits within it are marked as solved. Then the next unsolved output is found and the process repeats until finished.

The second step of the transformation is to delete the sets in which all elements have more than one solved mark. This gives something close to the ideal two-level array. Fitting the logic into a FPGA would require a third step to convert this ideal array into a multi-level array that would fit into their finer structure. This could be accomplished by recursively factoring gates from the high-level sets and ORing them together.

Conclusion

Forth provides a good foundation for a VHDL system because Forth is an extensible virtual interpreter. Most everyone who works with Forth knows its unique features can enhance software productivity. My experience has shown it to be very useful when working with variable hardware, as well.

The Forth inner interpreter is a very simple list processor that requires only three pointers, two registers, and an ALU to run efficiently. Because of Forth's simple structure, a software model can be completed very quickly, and it is easy to adapt it to changes in the instruction set as the design matures.

People who work with Forth have long known it is a good application language; our experience has shown that its advantages also apply when it used as a VHDL.

Bibliography

"VHDL and Verilog fundamentals- expressions, operands and operators."
Douglas J. Smith, *EDN*, 4/10/1997.

"VHDL & Verilog Syntax & Semantics Handbook."
Johan Sandstrom, *Integrated System Design Magazine*, Jan. 1996.

"Vhsic Hardware Description Language."
Steven H. Leibson, *EDN*, 3/16/1989.

"Getting a handle on HDLs."
Brian Dipert, *EDN*, 5/7/1998.

"Adopting VHDL for PLD design and simulation."
Troy Scott, *EDN*, 4/9/1998.

"Hug an XOR gate today: An introduction to Reed-Muller Logic."
Clive "Max" Maxfield, *EDN*, 3/1/1996.

Appendix A. CARMAP Word Set

Variables: (Items)

```
MAX:GLB:INPUTS
```

I/O Definitions:

```
IO-GROUP "name"

INPUT "name" [ START BITS ]

OUTPUT "name" [ START BITS CLOCK XORS TERMS
                FLIP USES USEX SEL SELX]
  [ OE PTCLOCK are Lattice-specific commands ]

BITS ( n — )
```

A word that defines the number of bits used in an `INPUT` or `OUTPUT`.

```
START ( n — )
```

A word used in conjunction with `BITS` that sets the starting bit number. If `START` is not specified, the first bit number will be zero.

```
CLOCK "name"
```

A word that defines the clock for registered outputs.

```
XORS ( n — )
```

A word that sets the maximum number of inputs to be tried in the XOR term.

```
TERMS ( n — )
```

A word that sets the maximum number of inputs to a logic block.

`FLIP ( m – )`
A mask that defines which output bits in the truth table will be inverted.

`USES ( m – ) "name"`
A bit mask that defines what bits are used by an output. A counter is a function where each output bit depends on all of the bits less than it. The `USES` mask is rotated to the position of the current output bit. The upper bits in the mask are rotated into the lower bits so they will be used in counting functions.

`USEX ( m – ) "name"`
A bit mask that defines what bits should be tried in an `XOR` function. This word is used in conjunction with `USES`, and the mask rotates the same as for `USES`.

`SEL ( m – ) "name"`
A bit mask that defines a set of bits in a fixed position that are used as a selector. This word is like `USES` but the mask does not rotate .

`SELX ( m – ) "name"`
A bit mask that defines what bits should be tried in an `XOR` function. The mask stays in a fixed position. This word is used in conjunction with `SEL`.

`OE ( – )`
A word that defines an output-enable term for a Lattice device.

`PTCLOCK ( – )`
A word that defines a clock term for a Lattice device.

`END-IO-GROUP`
A word that closes the I/O group.

Software Simulation Words

`INVERT ( d – d )`
The logical `NOT` of the bits in a word.

`MAP[ ( v – ) n`
A word that creates an associative memory structure similar to a `CASE` statement.

`MAP ( v a – )`
A word that inserts a token (*v*) and its associated value (*a*) into the `MAP` structure.

`]MAP ( a – )`
A word that inserts the default value (*v*) into the `MAP` structure, and finishes the mapping function.

`]: ( – a )`
A word that changes the state to compilation and returns the address of the start of the compiled string.

`;[`
A word that inserts a `next` into the compiled string and changes the state back to interpret.

`>> ( io – ) "label"`
The top element on the stack is moved to the input and output registers. (This word is used for design verification.)

`>>O ( o – ) "label"`
The top element on the stack is moved to the output register.

`>>X ( x – ) "label"`
The top element on the stack is moved to the don't-care register.

`>>OX ( d x – ) "label"`
The top element on the stack is moved to the don't-care register, and the next element is moved to the output register.

`O>> ( – o ) "label"`
The output register is copied to the stack.

`TRUTH-TABLE: ( io-group_ads – ) "simulation_word"`
Builds the truth table for a function, and solves the logic equations.

`MAKEMACS`
Solves all of the logic equations in a design.

Hardware Simulation Words

`INIT-LOGIC`
Must be done before defining nodes.

`NODE "name"`
Creates a single-bit, self-fetching variable called `%name`.

`NODES ( s n – ) "name"`
Creates a multiple-bit, self-fetching variable called `%name`.

`CLOCK "name"`
Creates a single-bit, self-fetching variable called `%%name`.

`UPDATE-STATE`
Updates the state of the outputs for all functions.

`EXECUTE-CLOCK`
Copies the state of the outputs to the inputs.

`SIMLDF`
The name of the simulation vocabulary.

Lattice-specific words for defining I/O pins

`CLKMAC ( n io-group_ads – ) "name" FORGET "io-group_name"`

`IOMAC ( n io-group_ads – ) "name" FORGET "io-group_name"`

`IMAC ( n io-group_ads – ) "name" FORGET "io-group_name"`

Reconfigurable Architecture Computation Engine

RACE™

Abstract

Because Forth's performance isn't compromised by a limited number of registers, it was the logical choice for a processor in currently available PLDs. In the design process for this project, Forth words were coded in the primitive set and used as a key benchmark. The processor was optimized by repeatedly modifying, compiling, and testing the model until it could execute Forth words at four MIPS and fit into the PLD with room left for the state machines needed by the application.

Forth was used to simulate the design.
Forth was used to define the hardware.
Forth was used to convert the design into logic equations.
Forth was used to fit the logic equations into the PLD.
Forth was used to route the PLD's internal connections.
Forth was used to verify the logic equations.
Forth was used to assemble the application code.
Forth was used as the metacompiler.

Introduction

The design process began by making a software simulation of a very simple Forth processor, called the *miniForth*. Getting the miniForth up and running was one of the easiest parts of the job: the simulation code for the 27 primitives needed to build the Forth kernel took only a few days to write and debug. The miniForth was the starting point in an evolutionary process that involved running the application on the simulator, finding bugs, and correcting shortcomings. The viability of this method was clearly evident when the prototype hardware booted up and said "OK" without a glitch.

Description

The RACE is a 16-bit RISC processor that will execute code at 25 to 50 MIPS, using currently available parts. It fits into an ispLSI1048 PLD with about one third of the device free for application-specific logic. In our application, the remaining macro cells were used for state machines to control timing and motor currents.

The RACE is a Harvard architecture machine with two memory spaces, one for code and the other for data. In the present configuration, the PC is twelve bits, so code space is limited to 8K bytes; and the IP is fifteen bits, making 64K bytes available for programs.

Code Space

Code space contains lists of code that define primitives and handle interrupts. Three different conditions can be selected to control branching in code space. PC branching takes one cycle, but the code after a branch executes so, in some cases, a null has to be placed there, making the branch take two cycles.

Data Space

Data space contains the stacks, programs, and data for the application. The first 24 locations are dedicated for system variables or pointers. The address for these variables can be loaded in one cycle. The loaded value can be used as an address or a constant. The return stack is assigned locations from 256–382 and the data stack is assigned locations from 384–510. Locations from 65K–128K are used for application-specific data. DRAM is available for applications needing a large memory space.

Forth Primitives

Most Forth primitives take from four to eight code words, so Forth runs about 4 MIPS. Code operators were devised so they could be combined to build efficient Forth primitives and make best use of the PLD's limited resources, so some things were done in unconventional ways. Functions like `AND`, `OR`, `1+`, `2*`, and `2/` are easy to do in one cycle, but `+` and `–` had to be broken into multiple parts. First, the operands are half-added using an `XOR` command that takes one cycle. Then a special command is executed four times to complete the function and propagate the carry through all 16 bits.

The `0BRANCH` primitive is built using a command that copies the jump address into the IP if the top of the stack is equal to zero. `NEXT` is done by a command that conditionally loads the IP, depending on the state of bit zero in the instruction. If the bit is zero, the instruction is a call, and the PC is loaded with the address of the nesting code. If the bit is one, the rest of the bits in the instruction are loaded into the PC.

The RACE has two interrupts, one for the timer and one for external events. The branch-on-interrupt is part of the next command. To maintain an interrupt latency of less than two microseconds, there can be no more than 128 clocks between `NEXT` commands.

Multiplication is done by adding and shifting, and division is done by subtracting and shifting; both take more time to execute than the maximum allowed interrupt latency, so a conditional `NEXT` command called `(LOOP?)` was created to allow interruptable loops. Words that use the `(LOOP?)` have two CFAs. The first points to the beginning of the code; the second points to the start of the code that is repeated. If RP6 is high, `(LOOP?)` reloads the PC with the CFA pointing to the start of the loop; if RP6 is low, the PC is incremented, and the code following the loop is executed.

Commands

The majority of the commands were made for building Forth primitives, but there are several application-specific commands for booting, accessing DRAM, loading the timer, doing I/O, loading code memory, and addressing local variables.

John R. Hart • Tempe. Arizona
jhart@testra.com • www.testra.com

Code words are divided into five fields: the control field, the accumulator field, the memory address field, the stack pointer field, and the register address field. There are two types of commands: deferred and immediate. Immediate commands execute on the next clock edge, while deferred commands execute on the second clock edge.

When writing to code space, the data to be written is in the TR, and the MA points to a memory location in data space that contains the address. After a write to code space, the state of the AC and the flags is indeterminate.

The Control field (CSu)

The CSu is five bits wide and it controls system timing, memory access, ALU modes, the operation of the flags, temporary register, and instruction pointer. Execution of the CSu is deferred until the next code word.

Flags

In addition and subtraction, the CRY is added to the nybles, and the overflow goes into the CRY. Also, there are commands that can set the CRY true or false, load it with AC0, load it with AC15, or swap it with the FLG. The FLG can be used as IP0 by IP+CRY>IP, loaded with IP15 by a IP*2>IP command, or loaded with TR0 when TR is copied into the IP.

The Temporary Register (TR)

The TR has several purposes. By convention, `NEXT` fetches the second element on the stack to TR. The TR can be loaded from the AC, or from memory, and it is the destination for the `AND` command. The TR can be rotated four bits to the right, or shifted four bits to the right with the high four bits coming from the low four bits of the AC. The TR can also be loaded with minus one. During a read from the boot EPROM, the low byte of TR is copied into the high byte.

The Instruction Pointer (IP)

The IP points at the word in data space that is being interpreted. It can be loaded with the contents of TR. When the TR is copied into the IP, TR0 is also copied into the FLG. The IP is incremented when a memory read directly follows an IP>MA command. The CRY can be added to the IP using an IP+CRY>IP command which uses the FLG as IP0. The IP can also be rotated left one bit through the FLG with an IP*2>IP command.

The Accumulator Field (ACu)

The ACu is a two-bit immediate command that controls the operation of the AC register (top of stack). There are four modes for AC commands that are set by the previous CSu, so there is a total of sixteen AC commands.

The accumulator (AC) is the destination for most arithmetic and logic commands and, by convention, the top of stack data is kept in the AC. The AC can be loaded from the TR or the MA. When the AC is loaded with the MA, the FLG is copied into AC0. The AC is the source of data for memory writes.

Logical commands execute in one cycle, but arithmetic commands take multiple cycles. The ALU is only four bits wide, so a sixteen-bit operation takes four cycles. Each cycle shifts the data left four bits at a time, so after four cycles the nybles are back where they started. Additions and subtractions are preceded by an XOR of the TR with the AC which performs a half add. The next four cycles complete the add, and propagate the carry.

The Memory Address field (MAu)

The MAu is a three-bit immediate command that selects one of seven different addresses to be loaded into the Memory Address Register (MA). The MA points to the location in data space that is used by a read or a write operation. The MA can be loaded with the contents of the IP, the SP, the RP, the AC, or a constant from 0–22. A simple form of indexing can be done using an RA_AC>MA command, which loads the MA with the contents of the AC, ORed with a constant from 0–22.

The Pointer field (PTu)

The PTu is a three-bit immediate command that controls incrementing, decrementing, and loading of the SP and the RP.

The Stack Pointer (SP) is six bits, with the upper bits pointing to a fixed location. The data stack range is 64 words from location 256–382. The SP can be incremented or decremented using an SP+1 or SP-1 command, or it can be initialized using an SP_XOR_AC>SP command. The SP is loaded into the MA by a SP>MA command.

The Return Pointer (RP) is six bits, with the upper bits pointing to a fixed location. The return stack range is 64 words from location 394–510. The RP can be incremented or decremented using an RP+1 or a RP-1 command, or it can be initialized using an RP_XOR_AC>RP command. The Rp is loaded into the MA by an RP>MA command. RP6 is used as the limit flag for the `(LOOP?)` command.

The register address field (RAu)

The RAu is a four-bit immediate command used for addressing local variables and loading small constants.

Loops

Loops in code space are performed by a `(LOOP?)` command, which is like a `NEXT` command, except the IP does not increment until RP6 equals zero. The pointer to code that uses the `(LOOP?)` instruction is followed by a pointer to code following the `DO`. The code in front of the `DO` needs to save the RP and load the loop index into the RP. If the loop index is larger than the size of the RP, (six bits) the incrementing of the larger part of the index must be done by code.

By convention, loops or lists of code should have no more than 120 clock cycles between a `NEXT` or `(LOOP?)` command, otherwise the accuracy of the timer may be compromised.

Interrupts

The interrupts are tested during `NEXT` and `(LOOP?)`. A timer interrupt causes the PC to branch to address 180h, and an external interrupt causes the PC to branch to address 80h. If one of the interrupts is true during `NEXT` or `(LOOP?)`, the execution vector is copied into TR. TR must be saved in memory in order to "return" from the interrupt.

The SP must have a valid pointer when `NEXT` or `(LOOP?)` executes, so the interrupting procedure will have a place to save registers, if necessary. When the interrupt process is finished and any used registers have been restored, `NEXT` is executed with an address in MA pointing at the saved execution vector.

Conclusion

Reconfigurable processors are computational devices with reprogrammable logic and data paths, and can be adapted to

the needs of an application. For many applications, reconfigurable hardware has a computation potential that is orders of magnitude faster than fixed hardware.

This project demonstrates that a practical, reconfigurable microcontroller can be built using a PLD. The main disadvantage of current PLDs for this application is the large number of pins connecting them to the memory. If as little as 2K of internal memory were available, 28 fewer I/O pins would be needed and the part could run much faster.

Bibliography

"The configurable processor draws near"
Stanley Yang, p63 *EE Times*, 10/19/1998.

"Speedy 8-Bit Microcontroller Crafts Virtual Peripherals"
Dave Bursky, p36 *Electronic Design*, 8/4/1997.

"Soft Computing Reconfigures Designer Options"
Jim Turley, p76 *Embedded Systems*, April 1997.

"From Code to Logic"
Man/Machine, p23 *OEM Magazine*, Dec/Jan 1996.

"Stalking the Chameleon Computer"
Murray Disman, p67 *OEM Magazine*, Dec/Jan 1996.

"Shattering the programmable-logic speed barrier"
Brian Dipert, *EDN*, 5/22/1997.

"CPLD/FPGA devices, tools lure PLD designers into faster, denser logic"
Mike Donlin, *Computer Design*, Nov 1995.

Appendix A. Instruction Word Format

If NEST is one, the word points to code;
if it is zero, the word points to another list.

Appendix B. Code Word Format

Appendix C. Internal Registers

AC	Accumulator	16 bits	(top of stack)
TR	Temporary Register	16 bits	
IP	Instruction Pointer	15 bits	
SP	Stack Pointer	6 bits	
RP	Return Pointer	6 bits	
MA	Memory Address	15 bits	
PC	Program Counter	12 bits	

Flags:

CRY	Carry Flag	(not borrow flag when subtracting)
FLG	Shift Flag	(memory address bit 0)
ACZ	AC is zero	(top of stack)

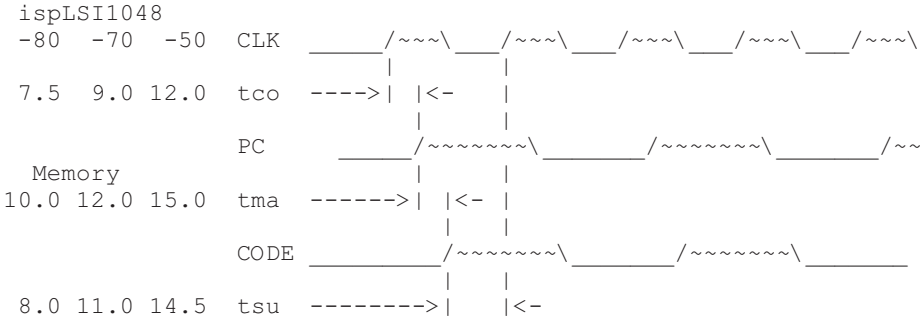
Appendix D. Code word set RACE4th Processor Version 7

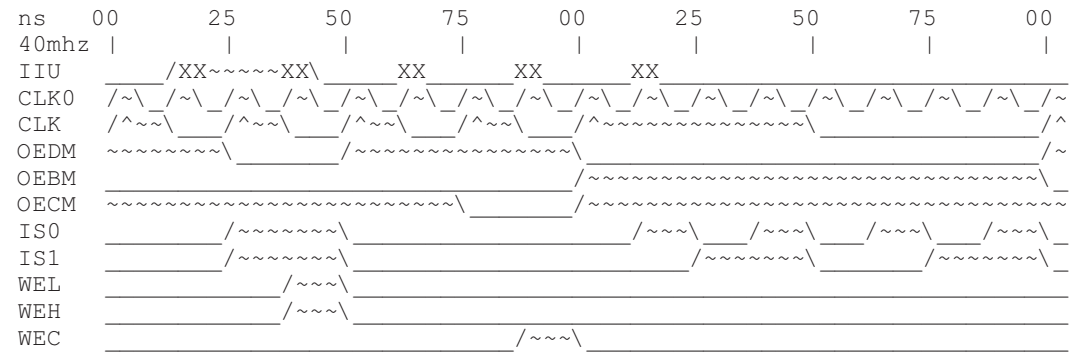
```
  Deferred Commands  (pipelined)
  5 CSu
      0  (nop)                        0>NM                          (Word Mode)
      1  (con)                        3>NM                     (Constant Mode)
      2  (>>TR)          >>TR         0>NM       (TR 4 bits right, Word Mode)
      3  (>>TR)          >>TR         1>NM     (TR 4 bits right, Nyble Mode)
      4  (-1>TR)         -1>TR                          (Minus one into TR)
      5  (AND>TR)        AC TR AND -> TR                (AC AND TR into TR)
      6  (?BRANCH)       IF_ZER TR>IP THEN    (IP branch if carry is false)
      7  (IF(C)>>TR)     IF_CRY >>TR THEN   (If CRY Rotate TR 4 bits Right)
      8  (AC>>TR)        AC>>TR 1>NM (Shift right low 4 bits of AC into TR)
      9  (@B)            BE  *MA_BM>TR                   (Read Boot EPROM)
                         IF_MA=IP  IP+2 THEN            (Increment the IP?)
      A  (!)             WE  AC>*MA  (AC is written into the memory at *MA)
      B  (@)             RE  *MA>TR                      (Read Data Memory)
                         IF_MA=IP  IP+2 THEN            (Increment the IP?)
      C  (AC>TR)         AC>TR                       (AC is copied into TR)
      D  (AC>TR>IP)      CALL          (The procedure address is in the TR)
      E  (!C)            IF_CRY  WEh ELSE  WEl THEN  (Byte Store Primitive)
      F  (NEXT)          IF_TINT      180>PC *MA>TR  ELSE
                           IF_INT      80>PC *MA>TR  ELSE
                             IF_MD0     0>PC *MA>TR  ELSE  *MA>PC THEN
                           THEN
                         THEN
                         IF_MA=IP  IP+2 THEN            (Increment the IP?)
     10  (>CRY)          f_RA>CRY           2>NM        (Word Mode, LD CRY)
     11  (NM)                               1>NM               (Nyble Mode)
     12  (>>TR)_(>CRY)   f_RA>CRY     >>TR  2>NM  (Word Mode, >>TR, LD CRY)
     13  (>>TR)          f_RA>CRY     >>TR  3>NM      (Constant Mode, >>TR)
     14  (JMPIF_CRY)     IF_CRY  (JMP) ELSE PC+2 THEN     (branch if Carry)
     15  (JMPIF/CRY)     IF_CRY  (JMP) ELSE PC+2 THEN (branch if Not Carry)
     16  (JMPIF_AC15)    AC15_IF (JMP) ELSE PC+2 THEN      (branch if AC15)
     17  (JMP)           *PC0..5_XOR_PC>PC                  (branch always)

     18  (IP+CRY)        IF_CRY          (Ads the Carry to the extended IP)
                           IF_FLG  0>FLG IP+2
                           ELSE    1>FLG
                           THEN
                         THEN
     19  (MA>TV)         Loads the Timer Register with MA1..9
     1A  (IP*2>IP)       Rotate IP left thru FLG
     1B  sp              Spare.
     1C  (!P)            DM>PC TR>*PC  80>PC    (Jumps to 80 when finished)
     1D  (BOOT)          DM>PC *BM>*CM 80>PC (copy 100 bytes from BM to CS)
     1E  (DRAM)          RA, 0=Clear, 1=RAS, 2=CAS       (RAS Disables RAM)
     1F  (LOOP?)         IF_RP6             (Loop with test for Interrupts)
                           IF_TINT       180>PC  *MA>TR  ELSE
                             IF_INT       80>PC  *MA>TR
                             ELSE        MA*>PC          THEN
                           THEN
                         ELSE            PC+2>PC         THEN


Memory Address:       For fetch and store.     ( used with @,  !, & NXT. )
   3 MAu
      0  IP             MA>MA                                      ( hold )
      1  IP             IP>MA     ( IP+2 if followed by @, IP1..15 -> MA )
      2  RP             RP>MA                        ( RP1..6 OR C0 -> MA )
      3  SP             SP>MA                        ( SP1..6 OR 80 -> MA )
      4  RG             RA>MA                        ( RA1..3       -> MA )
      5  RG             RA>MA                        ( RA1..3 OR 10 -> MA )
      6  RG             RA>MA                        ( RA1..3 OR 20 -> MA )
      7  IDX            RA_AC>MA                ( RA1..3 OR AC1..15 -> MA )
```

```
Arithmetic, Logic, Carry, and Flag.          ( previous CTL% sets the NM^ )
                                    ( CRY  uses RA0..1,  FLG uses RA0..2 )
 NMv ACu RAu
    0 0    Hold                 AC, FLG, and CRY, are not changed.
    0 1    AC_XOR_TR>AC         AC TR XOR -> AC
    0 2    MA>AC                MA -> AC
    0 3    TR>AC                TR -> AC

    1 0    AC+CRY>>AC           AC  CRY + -> AC   ACn4 -> CRY
    1 1    AC-/CRY>>AC          AC /CRY - -> AC  /ACn4 -> CRY
    1 2    AC+TR+CRY>>AC        AC TR XOR   TR +  CRY + -> AC  ACn4 -> CRY
    1 3    AC-TR-/CRY>>AC       AC TR XOR   TR - /CRY - -> AC /ACn4 -> CRY

    2 0 0  hold
    2 0 1  CRY>FLG              CRY -> FLG
    2 0 2  AC15.XOR.CRY>FLG     AC15 CRY XOR -> FLG
    2 0 3  FLG>AC_0>FLG         FLG -> AC  0 -> FLG
    2 0 7  FLG>AC_1>FLG         FLG -> AC  1 -> FLG

    2 1 0  0>CRY                0   -> CRY
    2 1 1  CRY><FLG             FLG -> CRY    CRY -> FLG
    2 1 2  1>CRY                1 -> CRY
    2 1 3  AC0>CRY              AC0 -> CRY

    2 2 ]: RAU>AC               RAU -> AC


    2 3 0  CRY>AC_0>CRY         CRY -> AC  0   -> CRY
    2 3 1  CRY>AC_FLG>CRY       CRY -> AC  FLG -> CRY
    2 3 2  CRY>AC_1>CRY         CRY -> AC  1   -> CRY
    2 3 3  CRY>AC_AC0>CRY       CRY -> AC  AC0 -> CRY

    3 0    CRY>AC               CRY         -> AC
    3 1    AC*2>AC              AC 2*       -> AC  AC15 -> CRY
    3 2    AC0_XOR_CRY>AC       AC0 CRY XOR -> AC
    3 3    AC*2+CRY>AC          AC 2* CRY + -> AC  AC15 -> CRY


Stack Pointers:                    ( Pointers are held when writing to CM )
     PTu
      0  nop             (hold)
      1  SP+2>SP         SP+2
      2  SP-2>SP         SP-2
      3  AC_XOR_SP>SP    AC SP XOR -> SP
      4  sp              (reserved)
      5  RP+2>RP         RP+2
      6  RP-2>RP         RP-2
      7  AC_XOR_RP>RP    AC RP XOR -> RP


Control Vectors:
Temporary Register.
  3  TRv
      0  TR>TR          Hold
      1  AND>TR         AC TR AND >> TR
      2  *MA>TR         Copy memory data into TR
      3  AC>TR          Copy AC into TR
      4  >>TR           Rotate TR right 4 bits
      5  AC>>TR         Shift TR right 4 bits. Copy AC_03 into TR_CF
      6  *MA_BM>TR      Copy TR0_7 into TR8_F and MD0_7 into TR0_7
      7  -1>TR          Load -1 into TR
Accumulator and Carry.
  2  NMv                Nyble Mode
      0  0>NM           Word Mode                                    (logic)
      1  1>NM           Nyble Mode                                    (math)
      2  2>NM           Word Mode, Load Carry                        (logic)
      3  3>NM           Constant Mode, Load Carry                 (Constant)
```

```
Program Counter.
  3  PCv
      0  PC+2          Increment  PC                          ( default )
      1  PC>PC         Hold                          ( state machine mode)
      2  *PC_X>PC      Jump Current Page                  ( XOR relative )
      3  *PC_X100>PC   Jump into Common Page                ( 100 - 17F )
      4  *MA>PC        MD0 IF 0>PC ELSE *MA>PC THEN              ( NEXT )
      5  0>PC          Jump to   0           (address of Code for nesting)                 ( 0 )
      6  80>PC         Jump to  80      (address of code for external int)
      7  180>PC        Jump to 180         (address of code for timer int)
Instruction Pointer.
  2  IPv
      0  IP>IP         Hold
      1  IP+2          Increment IP
      2  IP*2>IP       Shift left IP thru FLG
      3  TR>IP         Copy TR into IP
```

Appendix E. Timing






```
Counter timer:                      A 9 8 7 6 5 4 3 2 1
                                    | `-' `-----.-----'
  Start              ---------------'  |       |
  Chopper Mode.      ------------------'       |
   0  Measure                                  |
   1  Decide                                   |
   2  Settle                                   |
   3  Charge                                   |
  Load Value         ----------------------------'
   0  Off
   1  129 counts
  7F  256 counts

  1. A WTV command copies the contents of MA into the counter pre-load register.
  2. Counter is reloaded after it passes thru zero and the processor is interrupted. (addr 180h)
  3. The counter has it's own external clock but it must be in sync with the Processor clock.
```

Appendix F. Pins

Processor I/O:			
Data Memory	DM_0-15	(128K bytes)	16
Code Memory	CM_0-15	(8K bytes)	16
Outputs:			
Memory Address	MA_1-16		16
Pointer to Code	PC_1-12		12
I/O Command	IO_0-3		4
Chopper feedback	MEAS		1
Interrupt	INT		1
total			64
Micro Step I/O:			
Driver Outputs	PHA_0-7		8
	PHB_0-7		8
Comparator Inputs	CPI_0-7		8
	CNI_0-7		8
total			32
Misc. Inputs:			
Internal Clock	PCLK		1
Timer Clock	TCLK		1
Input Clock	ICLK		1
Output Enable TR	OETR		1
Output Enable AC	OEAC		1
Reset	RESET		1

Appendix G. Macro Cells & Logic Blocks

Counters:		cells	blks
Program Counter	PC	12	3
Instruction Pointer	IP	16	4
Return Pointer	RP	6	2
Stack Pointer	SP	6	2
Sub total		40	11
Address:			
Memory Address	MA	15	5
Data:			
Temporary Register	TR	16	4
Accumulator	AC	19	8
sub total		35	12
Control Logic:	CTL		
State	SQv	3	1
IP vector	IPv Instruction Pointer	2	
TR vector	TRv Temporary Register	3	1
II vector	IIv Input output	4	1
Flags	CRY FLG	2	1
Strobes	WTv	1	1
PC vector	PCv	3	1
NM vector	NMv	2	1
Sub total		21	7
Processor total		111	35
Driver:			
Timer		12	3
Chopper		24	8
spares		39	2
total		196	48

The Open Interpreter Word Set

Abstract

The concept of Open Interpreter makes the techniques of changing the control flow via return stack changes architecture-independent. The five classes of open interpreter systems allow programmers to choose the most adequate degree of compromise between portability and convenience of programming. The Open Interpreter specification presented in this paper may be used as an additional chapter to the ANSI/ISO standard.

1. The purpose of this paper

The purpose of this paper is to introduce a specification which would allow portable use of techniques that are currently (in March 1999) outside the scope of the ANS Forth standard. They are: manipulations with return addresses, backtracking, keeping literals in threaded code, user-defined control structures (ANS Forth supports the latter in a restricted way). Such techniques as user control over code generation, dynamic code generation, de-compilation will also benefit.

The value of some of the mentioned techniques is arguable, but, in fact, sufficient motivation is provided by the two following items:

1. portability of return address manipulations (which, in particular, means portability of backtracking);
2. portability of implementation techniques (in particular, of access to literals in threaded code). Portability of implementation techniques is valuable for cross-compilers and embedded systems: people often need to port a system to a new target keeping its internals the same.

To prevent possible misinterpretation, I have to expand on the second item. It is good when implementation tools are portable. They will not be as much portable as Core words, and the structure of the standard with the Open Interpreter specification reflects this: the code that e.g. accesses in-line data requires the system to support the Core word set, plus the optional Open Interpreter word set, plus the optional Open Interpreter In-line Data Access word set. It is up to the programmer to realize that some method is less portable than another, and to use it adequately. It is a bad style to mix low-level and application-level code, but a programming language standard cannot and must not prevent bad style.

The Open Interpreter word set will be proposed for inclusion into the standard, but first of all, the procedure requires this item be included into the technical committee (TC) agenda. It is possible that TC will not be willing to spend time on it. On the other hand, portability of the mentioned Forth techniques and inclusion of corresponding words into the standard are related, but different purposes. The proposed specification works even not being a part of the standard.

2. The approach

Let us formulate the main contradiction:

- the "classical" architecture is backed by a wide common practice, it is both simple and adequate to the techniques of return address manipulations, but there are also "unclassical" architectures, and therefore the code written for the "classical" model is not much portable;
- it is possible to write programs as if the return address size is unknown, the code will be portable, but cumbersome; this approach is not justified if the program will never be ported to a system with return addresses wider than one cell; in addition, double-cell return addresses are not widely used today;
- The compromise, "intermediate" solutions may be adequate for some architectures, but such compromises lose both advantages: they are neither backed by wide common practice nor widely portable.

The solution is to introduce multiple classes of Open Interpreter systems (namely, five). A "classical" system is of Class 1, and Class 5 is a probably Harvard system with probably multiple-cell return addresses and probably different size of code and data memory address units. A Class 1 system may be considered as a particular case of a Class 5 system.

The code written for higher classes may run on lower classes, but not vice versa. Therefore, programs written for higher classes are more portable. In exchange, programming for lower classes is less cumbersome (the word 'cumbersome' means 'inadequately complex').

Document: The proposed Open Interpreter Wordset.
History:
Version 1.0: Gassanenko M.L. *Open Interpreter: Portability of Return Stack Manipulations*. Proc. of the euroFORTH'98 Conf., Sept. 18–21 1998.
Version 2.0: Feb.–May 1999. Both a paper for *FD* and a proposal for the ANSI/ISO Forth standard.
Version 2.1: 29 June 1999. Added **`R-SAVE-SYS/R-RESTORE-SYS`**.
Version 2.2: 20 Oct 1999 Grammar corrections, better formulations, etc.

M.L. Gassanenko, Ph.D. • St. Petersburg, Russia
mlg@forth.org

M.L. Gassanenko is a researcher at the St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences.

An important requirement is that each lower class is a subclass of all higher classes. It guarantees that any Open Interpreter system belongs to one and only one minimal class. Otherwise it would be possible to consider the same system as a particular case of two different, incompatible architectures (each architecture implies the use of its own protocol, and the system would be able to implement either the first or the second protocol, but not both). Two portions of code written for the same system but assuming it to belong to different classes (that is, to implement different protocols) would be incompatible, which is absurd. Therefore, out of any two classes, one class must be a subclass of the other (one of the two protocols will have to include the other).

A system is said to be of Open Interpreter Class *N* if Class *N* provides the strongest specification the system can implement.

One more problem is related to stack manipulations:
- if the size of a return address is greater than one cell and unknown, too many stack operators are needed to manipulate with stack items of various sizes;
- programs that assume one-cell return addresses size are not much portable.

The solution is to use the return stack for data rearrangement. Return addresses come from the return stack and go to the return stack. In most cases, changes affect only two top elements. Therefore, the following set is enough:
>RR , move to the return stack;
RR> , move from the return stack;
RR@ , copy from the return stack;
RRDROP , remove from the return stack;
>RR< , exchange the data stack top with the return stack top;

3. The result

With the Open Interpreter specification, return address manipulations become portable across Open Interpreter systems. The five classes of open interpreter systems allow programmers to choose the most adequate degree of compromise between portability and convenience of programming.

Portability of return address manipulations enables one to use the following techniques to develop portable Forth code running on a variety of platforms:
1. user-defined (application-specific) methods of code execution, including backtracking;
2. data execution (data-driven approach);
3. user-defined (application-specific) control structures, including those for the techniques mentioned above;
4. access to parameters stored in threaded code via the return stack (it is a widely used and therefore important implementation technique).

Among application areas, we should mention distributed artificial intelligence and cross-compilers (tools for programming for embedded systems).

4. Document organization

The document contains references to sections of the ANS Forth Standard (ANSI X3.215-1994 American National Standard for Information Systems — Programming Languages — Forth, American National Standards Institute, Inc., 1994; also recognized as an ISO standard), for example, "1.3 Document organization" references the section 1.3 of the ANS Forth standard. The references to sections of the Open Interpreter word set specification all begin with OI, and sections of this paper that do not belong to the Open Interpreter specification are not referenced.

The glossary entries are organized according to ANS Forth rules (2.2.4 Glossary notation). The symbol ???? in the glossary entry number is used for words that do not have a sequential number assigned by the standard. The sequential numbers are a Technical Committee's prerogative. From similar considerations, the symbol OI (from Open Interpreter) is used in place of the section number.

OI The optional Open Interpreter word set

OI.1 Introduction

Since the very first implementation, Forth allowed access to the return addresses on the return stack. Nevertheless, it was not until the end of 1990s that the problem of portability of this technique was solved. The five classes of Open Interpreter systems allow programmers to choose the most adequate degree of compromise between portability and convenience of programming.

OI.2 Additional terms and notation

OI.2.1 Definition of terms and classes of Open Interpreter systems

OI.2.1.1 The five classes of Open Interpreter systems

Definition. There are 5 classes of open interpreter Forth systems:

Class 1. Return addresses have the same format as data addresses, the system uses threaded code which resides in data memory.

Class 2. Return addresses are 1 cell wide, but their representation on the return stack may be different from that on the data stack. Threaded code resides in data memory, and data stored into threaded code may be accessed by data memory access operators, such as @ . Both aligned and unaligned addresses may be converted to the return stack representation.

Class 3. Return addresses are 1 cell wide, their representation may be different from that of data addresses. Threaded code may reside in a separate memory, and special words may be required to access that memory. Both aligned and unaligned addresses may be converted to the return stack representation.

Class 4. Return addresses may be more than 1 cell wide, and special words may be required to access threaded code. Both aligned and unaligned addresses may be converted to the return stack representation. The size of a character is an integral multiple of the size of a code memory address unit.

Class 5. Return addresses may be more than 1 cell wide, and special words may be required to access threaded code. Conversion to the return stack representation is allowed only for compiled-token-aligned return addresses. The size of one code memory address unit may exceed the size of a character.

Each class is a subclass of the next class. (End of the

definition.)

A system is said to be of Open Interpreter Class *N* if Class *N* is the strongest specification the system can implement. (By definition of Open Interpreter Classes, if a system can implement the functionality of Class *N*, it also can implement the functionality of Class *N*+1—so classes with the smaller ordinal numbers have the stronger specifications.)

OI.2.1.2 Definition of terms

aligned code pointer: a code memory address at which a compiled token or a reference may be located.

cell-aligned code pointer: a code memory address at which a data cell may be located. Required to be the same as *aligned code pointer*.

code interpreter: the interpreter that processes threaded code, as specified in **OI.3.4 The executable code and the code interpreter**.

code memory address unit: the size of a code memory address unit may be different from that of a data memory address unit. See: *address unit* in **2.1 Definition of terms.**

code pointer: the address of a threaded code element (or, which is the same, the address of the threaded code fragment starting from that threaded code element).

compiled token: a threaded code element that denotes execution semantics of some procedure. When a compiled token is processed by the code interpreter, the corresponding execution semantics are performed. Different compiled tokens may have different sizes, but the ones generated by the word **TOKEN,** all have the same system-defined size.

current code fragment: The code fragment whose compilation has been started but not yet ended.

high-level definition: a definition created by the word **:** (colon) or by the **CREATE**...**DOES>** construct. The execution semantics of a high-level definition are implemented using threaded code.

in-line data (stored into threaded code): data stored into threaded code. The procedure whose compilation token precedes in-line data is responsible for processing these data. The procedure must also prevent processing of the in-line data by the code interpreter, for example, by advancing IP to the compiled token next to the data.

interpretation pointer (IP): the pointer to the next compiled token to be processed by the code interpreter. More precisely, the interpreter fetches the compiled token at IP, then advances IP to the next threaded code element, then executes the semantics denoted by the compiled token. See **OI.3.4 The executable code and the code interpreter**.

interpretation stack: the stack formed by IP (the top) and the return stack (the rest). The interpretation stack contains (1) code pointers that reflect the currently unfinished procedure calls, and (2) data that procedures place onto the return stack. The top interpretation stack element (IP) is always a code pointer.

IP: see *interpretation pointer.*

reference (to a threaded code fragment): a threaded code element that identifies the location of another threaded code element (and of the threaded code fragment starting from that element). The format of threaded code references is implementation-defined. This format may be used to represent the destination locations of control-flow operations.

return address: a code pointer which usually either a) is the run-time nesting information generated by the threaded code interpreter when a high-level definition is called; b) may be placed onto the return stack to let the code interpreter execute a code fragment; c) (rarely) is a code pointer which is, or could be, used as, or instead of a return address (in the sense of the *a)* and *b)* items).

threaded code: a) a sequence of threaded code elements; b) the representation of a program in the form of sequences of threaded code elements.

threaded code element: either a compiled token, a reference to a threaded code fragment, or in-line data.

threaded code fragment: a sequence of threaded code elements.

threaded code interpreter: the same as *code interpreter.*

unaligned code pointer: a code memory address, at which an in-line data element may be located. A compiled token and a reference may be located only at compiled-token-aligned addresses (aligned code pointers).

OI.2.2 Notation

OI.2.2.1 Interpretation stack notation

The interpretation stack notation is:

(I: *before* -- *after*)

The symbol "I:" is the interpretation stack *stack-id*. See 2.2.2 **Stack notation**.

Advancing IP to the next compiled token (**see OI.3.4 The executable code and the code interpreter**) is attributed to the threaded code interpreter and therefore is not included into the interpretation stack effect.

OI.2.2.2 Stored data notation

cp[<data>]	a code pointer *cp*, at which *<data>* are stored
cp+	the code pointer *cp* advanced by the size of data stored at *cp*
addr[<data>]	address *addr* at which *<data>* are stored

OI.3 Additional usage requirements

A system that provides either the Open Interpreter In-Line Data Access word set or the Open Interpreter Threaded Code Access word set shall provide the Open Interpreter word set.

OI.3.1 Data types

Append table OI.1 to table 3.1. Two different formats may be used to keep code pointers on the data stack and on the return stack. The data stack format is suitable for the read (or read/write) access to the code memory; the return stack format is suitable for the code interpreter.

Table OI.1 — Data Types

Symbol	Data type	Size on stack
acp-r	aligned code pointer (1)	depends on the system's class (3)
acp-s	aligned code pointer (2)	depends on the system's class (3)
ucp-r	unaligned code pointer (1)	depends on the system's class (3)
ucp-s	unaligned code pointer (2)	depends on the system's class (3)
acp	aligned code pointer (4,5)	depends on the system's class (3)
ucp	unaligned code pointer (4,5)	depends on the system's class (3)
cp	code pointer (6,5)	depends on the system's class (3)
ct	compiled token	none (size in code is implementation-defined)
ref	reference	none (size in code is implementation-defined)
*i*x* (7)	any data type	0 or more cells

(1) in the return stack representation
(2) in the data stack representation
(3) 1 cell (Classes 1-3); implementation-defined (Classes 4,5).
(4) the symbols *ucp* and *acp* denote, correspondingly, the types *ucp-s* and *acp-s* on the data stack diagrams and the data types *ucp-r* and *acp-r* on the return stack diagrams.
(5) When this symbol appears in both return stack and data stack diagrams suffixed with the same digit, it denotes the same value in the two representations. For example, the notation "(*cp* --) (R: -- *cp*) Move *cp* from the data stack to the return stack" means for Classes 1-4 "(*ucp-s* --) (R: -- *ucp-r*) Convert *ucp-s* to the return stack representation *ucp-r,* remove *ucp-s* from the data stack and place *ucp-r* onto the return stack".
(6) the symbol *cp* denotes the data type *ucp* for Classes 1-4 and the data type *acp* for Class 5.
(7) Like *i*x, j*x, k*x,* it may be an undetermined number of stack entries of unspecified type. See table 3.1.

Table OI.2 - Data Type Relationships

Open Interpreter data type	Class 1 data type	Class 2 data type	Class 3 data type	Class 4 data type	Class 5 data type
ucp-r	*=addr*	*=>x*	*=>x*	*unspecified*	*not exists*
acp-r	*=a-addr*	*=>ucp-r*	*=>ucp-r*	*=>ucp-r*	*unspecified*
ucp-s	*=addr*	*=>addr*	*=>u*	*=>i*x*	*=>i*x*
acp-s	*=a-addr*	*=>a-addr*	*=>ucp-s*	*=>ucp-s*	*=>ucp-s*
R: *ucp*	*=ucp-r*	*=ucp-r*	*=ucp-r*	*=ucp-r*	*not exists*
R: *acp*	*=acp-r*	*=acp-r*	*=acp-r*	*=acp-r*	*=acp-r*
S: *ucp*	*=ucp-s*	*=ucp-s*	*=ucp-s*	*=ucp-s*	*=ucp-s*
S: *acp*	*=acp-s*	*=acp-s*	*=acp-s*	*=acp-s*	*=acp-s*
R: *cp*	*=ucp-r*	*=ucp-r*	*=ucp-r*	*=ucp-r*	*=acp-r*
S: *cp*	*=ucp-s*	*=ucp-s*	*=ucp-s*	*=ucp-s*	*=acp-s*

OI.3.2 Data type relationships

The data type relationships for systems of different classes are given in table OI.2. The phrase "=> *j*" in the row corresponding to data type *i* denotes "*i* is a subtype of *j*", the phrase "= *j*" denotes "*i* is the same data type as *j*". The notation S: *i* indicates that the row describes the meaning of the data type symbol *i* on data stack diagrams; analogously, the notation R: *i* is used to describe the meaning of *i* on return stack diagrams.

See: **A.OI.3.2 Data type relationships**.

OI.3.3 Threaded code memory addresses

A code memory address identifies a location in the code memory space with a size of one code memory address unit, which a program may fetch from or store into or transfer control to except for the restrictions established in this Standard. The size of a code memory address unit is specified in bits. Each distinct code memory address value identifies exactly one such storage element.

The set of character-aligned code memory addresses, addresses at which a character can be accessed, is an implementation-defined subset of all code memory addresses. Adding the size of a character to a character-aligned code memory address shall produce another character-aligned code memory address.

The set of compiled-token-aligned (aligned) code memory addresses, addresses at which a compiled token or a reference can be accessed, is an implementation-defined subset of all code memory addresses. Adding the size of a reference or of a compiled token to a compiled-token-aligned address shall produce another compiled-token-aligned address. Code memory addresses (compiled-token-aligned, unaligned) are also called code pointers (aligned, unaligned).

Table OI.3 - Environmental Query Strings

String	Value data type	Constant?	Meaning
`OPEN-INTERP`	*flag*	*no*	Open Interpreter word set present
`OPEN-INTERP-EXT`	*flag*	*no*	Open Interpreter extensions word set present
`OI-DATA`	*flag*	*no*	Open Interpreter in-line data access word set present
`OI-DATA-EXT`	*flag*	*no*	Open Interpreter in-line data access extensions word set present
`OI-CODE`	*flag*	*no*	Open Interpreter threaded code access word set present
`OI-CODE-EXT`	*flag*	*no*	Open Interpreter threaded code access extensions word set present

The set of cell-aligned code memory addresses is an implementation-defined subset of character-aligned code memory addresses. The set of cell-aligned code memory addresses is the same as the set of compiled-token-aligned code memory addresses. Adding the size of a cell to a cell-aligned code memory address shall produce another cell-aligned code memory address.

Two representations are used for code pointers: the data stack format and the return stack one (for Class 1 systems they are the same). The return stack representation is the one used by the code interpreter, this format allows to execute code. The data stack representation permits address arithmetic and access to threaded code elements.

The code memory address units do not necessarily have the same size as data space address units. The size of a cell in data space address units may be different from the size of a cell in code memory address units.

The size of a reference and the size of a compiled token shall be integral multiples of the size of a code memory address unit.

OI.3.4 The executable code and the code interpreter

The executable code used by the Forth code interpreter is called *threaded code*. Threaded code is a sequence of *threaded code elements*, each one may be either:

- a *compiled token* of a procedure (that is, of a definition)
- a *reference* to threaded code (branch destinations are represented in this format)
- in-line data

Only compiled tokens of procedures are processed by the threaded code interpreter, the other two types of threaded code elements are processed by procedures. The procedure compiled immediately before in-line data and/or reference(s) shall modify IP to point to a valid compiled token, to prevent the code interpreter from accessing them.

The threaded code interpreter (the "inner" interpreter of Forth) has:

- a register (IP, the interpretation pointer) that points to the next threaded code element to be processed, and
- a stack (the return stack), to which the interpreter saves IP when it calls a threaded code fragment, and from which it loads IP exiting the threaded code fragment.

Together, IP and the return stack form the interpretation stack.

The threaded code interpreter repeats the following steps: fetches the compiled token at IP, then advances IP to the next threaded code element, then executes the semantics denoted by the compiled token. The semantics may imply changing IP. See **OI.6.1.0450 `:`** , **OI.6.1.0460 `;`** , **OI.6.1.1250 `DOES>`** , **OI.6.1.1380 `EXIT`** .

The interpretation stack can contain:

- code pointers that reflect the currently unfinished procedure calls, and
- data that procedures place onto the return stack.

The top interpretation stack element (IP) is always a code pointer.

Programs written for Open Interpreter Forth are allowed to change the number and order of interpretation stack elements. Programs written for Open Interpreter Forth are allowed to change control flow by changing the interpretation stack.

Programs are allowed to place data which are not threaded code fragment addresses onto the return stack, but these programs shall be written so that such data are never loaded into IP.

OI.3.5 Environmental queries

Append table OI.3 to table 3.5.

See: **3.2.6 Environmental queries**

OI.4 Additional documentation requirements

OI.4.1 System documentation

OI.4.1.1 Implementation-defined options

- class of the system;
- size and format of code pointers on the data stack and on the return stack;
- whether code space is a part of the data space, whether code is in a separate memory space;
- The method of converting from the data stack representation to the return stack representation (and vice versa);
- alignment requirements for threaded code elements;
- whether unaligned addresses may be correctly converted to the return stack representation;
- whether writing to code space is possible at run-time;
- environmental restrictions (if any) and additional disciplines they impose.

OI.4.1.2 Ambiguous conditions

- Loading IP with a value which is not a valid compiled token address in the return stack representation;

- compiling a word (adding corresponding semantics to the current definition) when the code memory pointer is not compiled-token-aligned;
- writing to code space at run-time;
- converting an unaligned code pointer to the return stack representation (Class 5 only);
- an unaligned code pointer is used where an aligned code pointer is required.

The following specific ambiguous conditions are noted in the glossary entries of the relevant words:

- the value passed to **OI.6.3.???? RP!** does not correspond to any valid return stack depth;
- **OI.6.3.???? RP!** removes from the return stack some data that control nesting structures, and the program does not restore these data (see: **OI.6.2.???? R-SAVE-SYS**, **OI.6.2.???? R-RESTORE-SYS**);
- an exception frame is removed by **OI.6.3.???? RP!**;
- word not defined via **6.1.1000 CREATE (OI.6.1.1250 DOES>);**
- *xt* passed to **OI.6.3.???? >TCODE** does not correspond to a colon definition;
- the destination address is unreachable (**OI.6.3.???? REF!**);
- *ct* has not been stored with **TOKEN**, or **TOKEN!** (**OI.6.3.???? TOKEN@**, **OI.6.3.???? TOKEN+**, **OI.6.3.???? TOKEN>**);
- the threaded code space pointer is not compiled token-aligned when **OI.6.5.???? /**, begins execution;
- on Class 5 systems, the code memory has not been allocated as a single cell (**OI.6.5.???? /@**);
- code memory address is not character-aligned (**OI.6.5.???? /C!**, **OI.6.5.???? /C@**);
- on Class 5 systems, the code memory at *ucp* has not been allocated as a single character (**OI.6.5.???? /C@**).

OI.4.1.3 Other system documentation

- the structure of executable code;
- how control structures are implemented;
- environmental restrictions, if any, and programming disciplines required in this connection.

OI.4.2 Program documentation

- the class of Open Interpreter required by the program;
- whether program writes to code memory at run-time;
- (optional) environmental restrictions which the system that runs the program is allowed to have.

OI.5 Compliance and labeling

Through the section **OI.5**, the symbol *wordset-name* denotes one of the following word sets: the Open Interpreter word set, the Open Interpreter Threaded Code Access word set, the Open Interpreter In-Line Data Access word set; the symbol *N* denotes the Open Interpreter class number of the system.

OI.5.1 ANS Forth systems

The phrase "Providing the *wordset-name* word set ***(specification ver. 2.2, proposed in <this publication>)***" shall be appended to the label of any Standard System that provides all of the *wordset-name* word set.

The phrase "Providing *name(s)* from the *wordset-name* extension word set ***(specification ver. 2.2, proposed in <this publication>)***" shall be appended to the label of any Standard System that provides portions of the *wordset-name* extension word set.

The phrase "Providing the *wordset-name* extension word set ***(specification ver. 2.2, proposed in <this publication>)***" shall be appended to the label of any Standard System that provides all of the *wordset-name* and *wordset-name* extension word set.

The phrase "Providing the *wordset-name* word set with environmental restrictions ***(specification ver. 2.2, proposed in <this publication>)***", or "Providing *name(s)* from the *wordset-name* extension word set with environmental restrictions ***(specification ver. 2.2, proposed in <this publication>)***", or "Providing the *wordset-name* extension word set with environmental restrictions ***(specification ver. 2.2, proposed in <this publication>)***" shall be appended to the label of any Standard System that provides names from the *wordset-name* [extension] word set, but imposes additional restrictions on their use.

The phrase "of Open Interpreter Class *N* ***(specification ver. 2.2, proposed in <this publication>)***" shall be appended to the label of any Standard System providing the Open Interpreter word set to indicate its Open Interpreter class.

OI.5.2 ANS Forth programs

The phrase "Requiring Open Interpreter Class *N* ***(specification ver. 2.2, proposed in <this publication>)***" shall be appended to the label of Standard Programs that assume the system to have the Open Interpreter class not higher than *N*.

The phrase "Requiring the *wordset-name* word set ***(specification ver. 2.2, proposed in <this publication>)***" shall be appended to the label of Standard Programs that require the system to provide the *wordset-name* word set.

The phrase "Requiring *name(s)* from the *wordset-name* Extension word set ***(specification ver. 2.2, proposed in <this publication>)***" shall be appended to the label of Standard Programs that require the system to provide portions of the *wordset-name* Extension word set.

The phrase "Requiring the *wordset-name* Extensions word sets ***(specification ver. 2.2, proposed in <this publication>)***" shall be appended to the label of Standard Programs that require the system to provide all of the *wordset-name* and *wordset-name* Extensions word sets.

OI.6 Glossary
OI.6.1 The Open Interpreter words

OI.6.1.0450 : "colon" OI

Replace the specification **6.1.0450 :** with the following one:
(C: "<spaces>name" -- colon-sys)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name*, called a "colon definition". Enter compilation state and start the current definition, producing *colon-sys*.

The execution semantics of name will be determined by

the words compiled into the body of the definition. The current definition shall not be findable in the dictionary until it is ended (or until the execution of **DOES>** in some systems). The code space pointer is aligned when **:** finishes execution.

name Initiation: (--) (I : *cp1* -- *cp1 acp2*)

Push the current value of IP onto the return stack and load IP with *acp2*, the address of the threaded code fragment in the *name*'s body, thus transferring control to the body of the definition.

name Execution: (*i*x* -- *j*x*) (I: *k*x cp1* -- *l*x acp3*)

Perform the initiation semantics of *name*. The rest of execution semantics, and the stack effects are due to the words compiled into the body of the definition. A compiled token must be located at the code memory address *acp3*. The symbols *i*x* and *j*x* represent arguments to and results from *name*, respectively. The symbols *k*x* and *l*x* represent changes on the return stack.

Note. If the optional Locals word set is present, the elements of the return stack are unavailable after declaration of locals. Nevertheless, after declaration of locals the top return stack element shall be an address to which **EXIT** may transfer control.

See: **6.1.0450 :**, **A.OI.6.1.0450 :**, ***RFI 0005 Initiation semantics.***

OI.6.1.0460 ; "semicolon" OI

Replace the specification **6.1.0460** *;* with the following one:

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *colon-sys* --)

Append the run-time semantics below to the current definition. End the current definition, allow it to be found in the dictionary and enter interpretation state, consuming *colon-sys*. If the data-space pointer is not aligned, reserve enough data space to align it.

Run-time: (--) (I : *acp1 cp2* -- *acp1*)

Transfer control to the code fragment specified by *acp1*.

See: **6.1.0460** *;* , **A.OI.6.1.0460** *;* , **OI.6.1.0450 :** , **OI.6.1.1380 EXIT .**

OI.6.1.???? >RR "to-double-r" OI
(*cp* --) (R: -- *cp*)
Move *cp* from the data stack to the return stack, converting it to the return stack format. On Class 1 systems, **>RR** is equivalent to **>R** .

OI.6.1.???? >RR< "to-double-r-and-back" OI
(*cp1* -- *cp2*) (R: *cp2* -- *cp1*)
Exchange *cp1* at the data stack top with *cp2* at the return stack top, changing their representation. For Class 1 - Class 3 systems, **>RR<** is equivalent to **RR> SWAP >RR** .

OI.6.1.1250 DOES> "does" OI

Replace the specification **6.1.1250 DOES>** with the following:

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *colon-sys1* -- *colon-sys2*)

Append the run-time semantics below to the current definition. Whether or not the current definition is rendered findable in the dictionary by the compilation of **DOES>** is implementation defined. Consume *colon-sys1* and produce *colon-sys2*. Append the initiation semantics given below to the current definition.

Run-time: (--) (I: *acp1 cp2* -- *acp1*)

Replace the execution semantics of the most recent definition, referred to as *name*, with the *name* execution semantics given below. Transfer (return) control to the (calling) threaded code fragment specified by *acp1*. An ambiguous condition exists if *name* was not defined with **CREATE** or a user-defined word that calls **CREATE** .

name Initiation: (-- *a-addr*) (I : *cp3* -- *cp3 acp4*)

Place *name*'s data field address on the stack. Push the current value of IP onto the return stack and load IP with *acp4*, the address of the threaded code fragment that follows **DOES>** which modified *name*, thus transferring control to the **DOES>** part of that definition.

name Execution: (*i*x* -- *j*x*) (I: *k*x cp3* -- *l*x acp5*)

Perform the initiation semantics of *name*. The rest of execution semantics, and the stack effects are due to the words compiled after the **DOES>** which modified *name*. At the code memory address *acp5* a compiled token must be located. The symbols *i*x* and *j*x* represent arguments to and results from *name*, respectively. The symbols *k*x* and *l*x* represent changes on the return stack.

See: **A.6.1.1250 DOES>** , **OI.6.1.0450 :**, **A.OI.6.1.0450 :** , **RFI 0003** ***Defining words etc., RFI 0005 Initiation semantics.***

OI.6.1.1370 EXECUTE CORE
(*i*x xt* -- *j*x*) (I: *k*x* -- *l*x*)
Remove *xt* from the stack and perform the semantics identified by it. Other stack effects are due to the word **EXECUTE**d. The stack effect of the executed word is assumed to be:
(*i*x* -- *j*x*) (I: *k*x* -- *l*x*)

See: **6.1.1370 EXECUTE, OI.6.1.???? RUSH.**

OI.6.1.1380 EXIT OI
(I: *acp1 cp2* -- *acp1*)
Replace the specification **6.1.1380 EXIT** with the following:

Transfer control to the code fragment specified by *acp1*.

OI.6.1.???? RR> "double-r-from" OI
(-- *cp*) (R: *cp* --)
Move *cp* from the return stack to the data stack, converting it to the data stack format. On Class 1 systems, **RR>** is equivalent to **R>** .

OI.6.1.???? RR@ "double-r-fetch" OI
(-- *cp*) (R: *cp* -- *cp*)
Copy *cp* from the return stack top to the data stack, converting it to the data stack format for code pointers. On Class 1 systems, **RR@** is equivalent to **R@** .

OI.6.1.???? RRDROP "double-r-drop" OI
(--) (R: *cp* --)
Remove *cp* from the return stack. On Class 1 - Class 3 systems, **RRDROP** is equivalent to **R> DROP** .

OI.6.1.???? RUSH OI
(*i*x xt* -- *j*x*) (I: *k*x cp1* -- *l*x*)
Remove the top interpretation stack element *cp1*, and then execute *xt*, that is, remove *xt* from the stack and perform the semantics identified by it, as with **EXECUTE** . Other stack effects are due to the word executed. The stack effect of executed *xt* is assumed to be:
(*i*x* -- *j*x*) (I: *k*x* -- *l*x*)

See **OI.6.1.1380 EXIT**, **6.1.1370 EXECUTE**, **OI.6.1.1370 EXECUTE**, **A.OI.6.1.???? RUSH.**

OI.6.2 The open interpreter extension words

OI.6.2.???? R-RESTORE-SYS "r-restore-sys" OI-EXT
(--) (R: *xn ... x1 n* --)
Restore implementation-dependent data *xn ... x1* about enclosing structures.

See: **A.OI.6.2.???? R-SAVE-SYS**, **OI.6.2.???? R-SAVE-SYS**, **OI.6.2.???? RP! .**

OI.6.2.???? R-SAVE-SYS "r-save-sys" OI-EXT
(--) (R: -- *xn ... x1 n*)
Save implementation-dependent data on the return stack. These data contain information about enclosing structures which (information) may be lost when a non-local exit is performed with the help of **RP!**. This information about enclosing structures (more precisely, the system copy of this information) does not change when a threaded code fragment is called or exited, or when values are placed onto or removed from the return stack.

See: **A.OI.6.2.???? R-SAVE-SYS**, **OI.6.2.???? R-RESTORE-SYS**, **OI.6.2.???? RP!**, **OI.6.2.???? RP@.**

OI.6.2.???? COPY>RR "copy-to-double-r" OI
(*cp* -- *cp*) (R: -- *cp*)
Copy *cp* from the data stack to the return stack, converting the copy to the return stack format. For Class1 - Class3 systems, **COPY>RR** is equivalent to **DUP >RR** .

OI.6.2.???? RADDR@ "r-addr-fetch" OI-EXT
(*a-addr* -- *cp*)
Fetch the code pointer *cp* stored at *a-addr*. For systems of Classes 1-3 this word is equivalent to **@** .

OI.6.2.???? RADDR! "r-addr-store" OI-EXT
(*cp a-addr* --)
Store the return address *cp* at *a-addr*. For systems of Classes 1-3 this word is equivalent to **!** .

OI.6.2.???? RADDR+ "r-addr-plus" OI-EXT
(*addr1* -- *addr2*)
Add the size in address units of a code pointer to *addr1*, giving *addr2*. For systems of Classes 1-3 this word is equivalent to **CELL+** .

OI.6.2.???? RADDR– "r-addr-minus" OI-EXT
(*addr1* -- *addr2*)
Subtract the size in address units of a code pointer from *addr1*, giving *addr2*. For systems of Classes 1-3 this word is equivalent to the phrase **1 CELLS -** .

OI.6.2.???? RP@ "r-p-fetch" OI-EXT
(-- *x*)
Return a system-dependent value identifying the current depth of the return stack. A Standard program may pass this value to OI.6.2.???? **RP!** or compare for equality to another such value.

OI.6.2.???? RP! "r-p-store" OI-EXT
(*x1* --) (R: *i*x* -- *j*x*)
Set the return stack depth to be the one specified by *x1*. If the new stack depth is greater than the old stack depth, the contents of the newly allocated return stack elements are undefined. An ambiguous condition exists if *x1* does not correspond to any valid return stack depth. An ambiguous condition exists if the return stack contains data that control nesting structures and the program does not restore such data. An ambiguous condition exists if an exception frame is removed by **RP!**.

See: **OI.6.2.???? R-SAVE-SYS**, **A.OI.6.2.???? R-SAVE-SYS**, **OI.6.2 . ???? R-RESTORE-SYS**, **OI.6.2.???? RP@.**

OI.6.3 The Open Interpreter threaded code access words

OI.6.3.???? /ALLOT "slash-allot" OI-CODE
(*n* --)
Calculate *m*, the amount of code memory address units enough to store *n* data memory address units. If *m* is greater than zero, reserve *m* code memory address units. If *m* is less than zero, release |*m*| address units of code space. If *m* is zero, leave the code-space pointer unchanged. If the code-space pointer is aligned and *n* is a multiple of the size of a compiled token or of a reference when **/ALLOT** begins execution, it will remain aligned when **/ALLOT** finishes execution.

See **OI.6.5.???? /ALLOT .**

OI.6.3.???? /HERE "slash-here" OI-CODE
(-- *ucp*)
ucp is the code memory space pointer.

OI.6.3.???? >TCODE "to-t-code" OI-CODE
(*xt* -- *acp*)

Return the address *acp* of the threaded code fragment which is called when the colon definition identified by *xt* is executed. An ambiguous condition exists if *xt* does not correspond to a colon definition.

OI.6.3.???? REF! "ref-store" OI-CODE
(*acp1 acp2* --)
Store a reference to *acp1* at *acp2*. After execution of this word, the reference at *acp2* points to *acp1*. The size of the modified code memory area may be calculated with the phrase **1 REFS** . An ambiguous condition exists if the destination address is unreachable. The address at which the reference is located and the address that follows it shall be always reachable.

OI.6.3.???? REF+ "ref-plus" OI-CODE
(*acp1[ref.acp2]* -- *acp1+*)
Advance *acp1* by the size of a reference.

OI.6.3.???? REF– "ref-minus" OI-CODE
(*acp1* -- *acp2*)
Decrease *acp1* by the size of a reference.

OI.6.3.???? REF@ "ref-fetch" OI-CODE
(*acp1[ref.acp2]* -- *acp2*)
Return *acp2,* the address to which the reference at *acp1* points.

OI.6.3.???? REFS OI-CODE
(*n1* -- *n2*)
n2 is the size in data space address units of *n1* references.

OI.6.3.???? TOKEN! "token-store" OI-CODE
(*xt acp* --)
Store a compiled token of the procedure identified by *xt* to the threaded code element located at *acp*. The compiled token may be retrieved by the word **TOKEN@** or executed with the code interpreter. The size of the modified code memory area may be calculated with the phrase **1 TOKENS** .

OI.6.3.???? TOKEN, "token-comma" OI-CODE
(*xt* --)
Add a compiled token of the procedure identified by *xt* to the current threaded code fragment. The compiled token may be executed with the code interpreter, or retrieved with the word **TOKEN@** , or changed with the word **TOKEN!** . The size of the added compiled token may be calculated by the phrase **1 TOKENS** .

OI.6.3.???? TOKEN@ "token-fetch" OI-CODE
(*acp[ct]* -- *xt*)
Decode the compiled token *ct* at *acp* and return the execution token *xt* of the procedure which semantics (compilation token *ct*) is stored at *acp*. An ambiguous condition exists if *ct* has not been stored there with **TOKEN,** or **TOKEN!** .

OI.6.3.???? TOKEN+ "token-plus" OI-CODE
(*acp[ct]* -- *acp+*)
Increment *acp* by the size of the compiled token *ct* at *acp*, returning the address of the next threaded code element. An ambiguous condition exists if *ct* has not been stored at *acp* with **TOKEN,** or **TOKEN!** .

OI.6.3.???? TOKEN> "token-from" OI-CODE
(acp[ct] -- *acp+ xt)*
Decode the compiled token at *acp* and return the address of the next threaded code element *acp+*, and the execution token *xt* of the procedure whose compiled token *ct* is stored at *acp*. An ambiguous condition exists if *ct* has not been stored at *acp* with **TOKEN,** or **TOKEN!**. The word **TOKEN>** is equivalent to the phrase **>RR RR@ TOKEN+ RR> TOKEN@**.

OI.6.3.???? TOKENS OI-CODE
(n1 -- *n2)*
n2 is the size in data space address units of *n1* compiled tokens allocated with the word **TOKEN,** .

OI.6.4 The Open Interpreter threaded code access extension words

None.

OI.6.5 The Open Interpreter in-line data access words

OI.6.5.???? /! "slash-store" OI-INLINE
(x acp -- *)*
Store one-cell data *x* at *acp*. On Class 1 systems, this word is equivalent to the word **!** .

OI.6.5.???? /+ "slash-plus" OI-INLINE
(n ucp1 -- *ucp2)*
Calculate *m,* the amount of code memory address units enough to store *n* data memory address units. Add *m* to *ucp1*. For systems of Classes 1 and 2 this word is equivalent to **+** .

OI.6.5.???? /, "slash-comma" OI-INLINE
(*x* --)
Reserve one cell of threaded code space and store x in the cell. If the threaded code space pointer is compiled token-aligned when **/,** begins execution, it will remain compiled token-aligned when **/,** finishes execution. An ambiguous condition exists if the threaded code space pointer is not compiled token-aligned when **/,** begins execution.

OI.6.5.???? /@ "slash-fetch" OI-INLINE
(*acp[x]* -- *x*)
Fetch the one-cell literal data *x* located at *acp*. On Class 1 systems, this word is equivalent to the word **@** . On Class 5 systems, an ambiguous condition exists if the code memory at *ucp* has not been allocated as a single cell.

If return addresses are one-cell wide and code memory is data memory, and if alignment requirements for compiled tokens and data memory cells are different (that is, aligned code pointers are not aligned addresses), the system can implement only Class 3.

OI.6.5.???? /ALIGN "slash-align" OI-INLINE
(--)
If the code-space pointer is not aligned, reserve enough space to align it.

OI.6.5.???? /ALIGNED "slash-aligned" OI-INLINE
(*ucp* -- *acp*)
acp is the first aligned code pointer greater than or equal to *ucp*.

OI.6.5.???? /ALLOT "slash-allot" OI-INLINE
Calculate *m,* the amount of code memory address units enough to store *n* data memory address units. If *m* is greater than zero, reserve *m* code memory address units. If *m* is less than zero, release |*m*| address units of code space. If *m* is zero, leave the code-space pointer unchanged. If the code-space pointer is aligned and *n* is a multiple of the size of a cell when **/ALLOT** begins execution, it will remain aligned when **/ALLOT** finishes execution. If the code-space pointer is character aligned and *n* is a multiple of the size of a character when **/ALLOT** begins execution, it will remain character aligned when **/ALLOT** finishes execution.

See **OI.6.3.???? /ALLOT**, **A.OI.6.5.???? /ALLOT**.

OI.6.5.???? /C! "slash-c-store" OI-INLINE
(*c ucp* --)
Store character *c* at *ucp*. When character size is smaller than cell size, only the number of low-order bits corresponding to character size are transferred. On Class 1 systems, this word is equivalent to the word **C!** . An ambiguous condition exists if *ucp* is not character-aligned.

OI.6.5.???? /C@ "slash-c-fetch" OI-INLINE
(*ucp[c]* -- *c*)
Fetch the character literal data located at *ucp*. An ambiguous condition exists if *ucp* is not character-aligned. For Class 1 systems, this word is equivalent to **C@** . On Class 5 systems, an ambiguous condition exists if the code memory at *ucp* has not been allocated as a single character.

OI.6.5.???? /CELL+ "slash-cell-plus" OI-INLINE
(*ucp1* -- *ucp2*)
Advance *ucp1* by the size of a cell. For Class1 and Class 2 systems, this word is equivalent to **CELL+** .

OI.6.5.???? /C, "slash-c-comma" OI-INLINE
(*char* --)
Reserve space for one character in the threaded code space and store *char* in the space.

OI.6.5.???? /GET "slash-get" OI-INLINE
(*addr u ucp* --)
If *u* is greater than 0, fill the *u* data space address units at *addr* with the contents of the corresponding amount of consecutive threaded code space address units at *ucp*. For systems of Classes 1 and 2, this word is equivalent to the phrase **ROT ROT MOVE** .

OI.6.5.???? /HERE "slash-here" OI-INLINE
See **OI.6.3.???? /HERE .**

OI.6.5.???? /PUT "slash-put" OI-INLINE
(*addr u ucp* --)
Calculate *m,* the amount of code memory address units enough to store *n* data memory address units. Fill the *m* code memory address units with the contents of *n* data memory address units at *addr.*

OI.6.6 The Open Interpreter in-line data access extension words

OI.6.6.???? //SWAP "double-slash-swap" OI EXT
(ucp1 ucp2 -- ucp2 ucp1)
Exchange *ucp1* and *ucp2* . For Class 1 - Class 3 systems, this word is equivalent to **SWAP** . For Class 1 - Class 4 systems, this word is equivalent to **>RR >RR< RR>** .

OI.6.6.???? /XSWAP "slash-x-swap" OI EXT
(ucp x -- x ucp)
Exchange *ucp* and *x* (*x* is at the stack top). For Class 1 - Class 3 systems, this word is equivalent to **SWAP** .

OI.6.6.???? X/SWAP "x-slash-swap" OI EXT
(x ucp -- ucp x)
Exchange *x* and *ucp* (*ucp* is at the stack top). For Class 1 - Class 3 systems, this word is equivalent to **SWAP** .

A.OI The optional Open Interpreter Wordset

A.OI.1 Introduction

A.OI.2 Additional terms and notation
A.OI.2.1 Definition of terms and classes of Open Interpreter systems
A.OI.2.1.1 The five classes of Open Interpreter systems

Class 5. It is possible that real Class 5 systems will not be able to support the Open Interpreter Data Access and Code Access word sets without environmental restrictions. If hardware does not permit unaligned code pointers to be stored as return addresses, the executable code memory space is most likely larger than the readable code memory space. For example, code memory may consist of 64K 16-bit words, but only the first 32K words may be accessed as 64K read-only bytes. In this situation, the full implementation of the code memory data access functionality is just not possible.

This wordset does not attempt to fully support the Class 5 systems; instead, it suggests an approach that enables the programmer, given a Class 5 system with some kind of environmental restrictions, to develop a wordset that will work both on Class 5 systems with this kind of restrictions and on systems of lower classes.

In general, it may be recommended to write new code for at least Class 3; Class 5 is probably not worth care unless there is a perspective of porting code to a Class 5 system.

A.OI.2.1.2 Definition of terms
A.OI.2.2 Notation
A.OI.2.2.1 Interpretation stack notation

A word having the return stack effect

(R: *i*x* -- *j*x*)

is assumed to have the following interpretation stack effect:

(I: *i*x cp* -- *j*x cp*)

and vice versa, only words that do not change IP, the top

interpretation stack item, may be adequately described by the return stack diagram.

A.OI.2.2.2 Stored data notation
A.OI.3 Additional usage requirements
A.OI.3.1 Data types

The data type *cp* denotes an unaligned code pointer (*ucp*) for Classes 1-4, and an aligned code pointer (*acp*) for Class 5, because unaligned code pointers cannot be represented in the return stack representation on systems of Class 5.

The data types *acp* and *ucp* have two representations: the data stack one (*acp-s* and *ucp-s*, correspondingly) and the return stack one (*acp-r* and *ucp-r*). The symbol *acp* denotes *acp-s* on the data stack diagrams and *acp-r* on the return stack diagrams. This approach has been chosen because *acp-s* and *acp-r* (*ucp-s* and *ucp-r*) are logically the same value.

A.OI.3.2 Data type relationships

For Class 1,
acp = a-addr => cp = ucp = addr.

For Class 2, the return stack representation of code pointers is different from the data stack representation.
acp-s = a-addr => cp-s = ucp-s = addr,
acp-r => cp-r = ucp-r => x.

For Class 3, the code pointers are not necessarily data memory addresses.
acp-s => cp-s = ucp-s => u,
acp-r => cp-r = ucp-r => x.

For Class 4, the code pointers are not necessarily one-cell wide.
*acp-s => cp-s = ucp-s => i*x,*
acp-r => cp-r = ucp-r.

A system of Class 5 is a system of Class 4 with the environmental restriction that unaligned code pointers cannot be converted to the return stack representation. This restriction affects all words that accept or return the data type *cp*.
*cp-s = acp-s => ucp-s => i*x,*
cp-r = acp-r => ucp-r.

The symbols *cp-s* and *cp-r* above denote *cp* in the data stack and return stack representations correspondingly.

A.OI.3.3 Threaded code memory addresses

The standard does not require that the size (in bits) of one code memory address unit is not greater than the size of a character, but it is possible that systems on which it is not true will not be able to implement the Open Interpreter In-Line Data Access word set in a reasonably efficient way.

A.OI.3.4 The executable code and the code interpreter

The key to understanding the return address manipulations is a dual view on the interpreter stack. The threaded code interpreter considers the return stack and the interpretation pointer (IP) as a single stack. The programmer manipulates only with the return stack, because IP changes while the programmer's code executes, and writing to IP will result in an immediate control transfer. On the other hand, this does not make a restriction: when we call an auxiliary procedure, the return stack becomes what the interpretation stack was. Any changes that have to be done with the interpretation stack, the auxiliary procedure does with the return stack. When the procedure exits, the interpretation stack becomes what the return stack was.

The rule of thumb for writing code that changes the interpretation stack is: write code that does with the return stack what must be done with the interpretation stack; put this code into an auxiliary procedure. This procedure will do the required changes with the interpretation stack.

A.OI.3.5 Environmental queries
A.OI.4 Additional documentation requirements
A.OI.4.1 System documentation
A.OI.4.1.1 Implementation-defined options
A.OI.4.2 Program documentation

A program written for a standard system with environmental restrictions can run on a standard system. A standard system provides all of the functionality of the system with environmental restrictions, plus some additional functionality.

A program written for an unstandard system cannot run on a standard system. The functionality of a standard system is just different from the functionality of the unstandard system for which the program is written.

For example, if the system does not implement the word **`/C@`** , it is an environmental restriction. If the system allows in-line literal data only within the first 32K of code memory, it is an environmental restriction. A program aware of such peculiarities still can run on a standard system. But if the value returned by **`RR@`** points to the called compiled token instead of the next compiled token, the system is unstandard, and a program aware of this peculiarity cannot run on a standard system.

A.OI.5 Compliance and labeling
A.OI.5.1 ANS Forth systems
A.OI.5.2 ANS Forth programs
A.OI.6 Glossary
A.OI.6.1 The Open Interpreter words

A.OI.6.1.0450 : “colon” OI

name Execution: (*i*x -- j*x*) (I: *k*x cp1 -- l*x acp3*)

1)
The initiation semantics of *name* has the interpretation stack effect
(I : *cp1 -- cp1 acp2*).
The rest of execution semantics of *name* has the interpretation stack effect
(I: *k*x cp1 acp2 -- l*x acp3*),
thus giving
(I: *k*x cp1 -- l*x acp3*).

2)
cp1 is not necessarily aligned because the word **OI.6.1.???? RUSH** enables one to start execution of a colon definition with an unaligned *cp1*. But if *name* is invoked by the threaded code interpreter, *cp1* just cannot be unaligned.

3)
If *name* does not do return stack manipulations, its interpretation stack effect
(I: *acp1* -- *acp1*)
is a "sum" of the interpretation stack effects of:
name initiation
(I : *acp1* -- *acp1 acp2*),
name body (IP changes while the body is being interpreted)
(I : *acp1 acp2* -- *acp1 acp4*),
and **EXIT** (or run-time semantics of **OI.6.1.0460 ;**)
(I : *acp1 acp4* -- *acp1*).

A.OI.6.1.???? RUSH OI

The word **RUSH** allows to get rid of an extraneous return stack element. If the word **X** does return stack manipulations, then the interpretation stack elements are arguments to it. Execution of **X** from inside an auxiliary definition is different from execution of **X** without an auxiliary definition because one more return address makes the difference. The word **RUSH** allows an auxiliary definition to call **X** as if **X** was called in place of the auxiliary definition.

A.OI.6.2 The open interpreter extension words

A.OI.6.2.???? RP@ "r-p-fetch" OI-EXT
A.OI.6.2.???? RP! "r-p-store" OI-EXT

The purpose of these words is to provide non-local exits (which is required, for example, for a Prolog-like *cut* statement). These words may be found on most (if not all) Forth systems.

The value *x* used by these words is traditionally called "return stack pointer".

OI.6.2.???? R-SAVE-SYS "r-save-sys" OI-EXT

An implementation may keep data that control nesting structures in registers. For example, it may keep in registers do-loop parameters (count, limit) and the locals frame pointer (if not locals themselves). Therefore, a program that implements non-local exits using **RP!** shall save such information using **R-SAVE-SYS** before obtaining the return stack pointer with **RP@** and shall restore this information using **R-RESTORE-SYS** after changing the return stack pointer with **RP!**.

OI.6.3 The Open Interpreter threaded code access words

There is a wide class of applications that do not need dynamic code generation or run-time patching of generated code. Therefore, it may be quite reasonable to introduce environmental restrictions on the use of words that write to code space, for example, requiring that these words are not used to patch finished definitions. Such system shall be labeled as "Providing the Open Interpreter threaded code access word set with environmental restrictions", and the restrictions shall be documented.

A.OI.6.3.???? TOKEN, "token-comma" OI-CODE

The difference between **OI.6.3.???? TOKEN**, and **6.2.0945 COMPILE**, is that **COMPILE,** is allowed to do optimizations. If some word, say **TUCK** , is compiled with **TOKEN,** , the resulting compiled token is guaranteed to have the size of **1 TOKENS** and be decompiled (e.g., with the word **TOKEN>**) as **TUCK** , while if the same word is compiled with **COMPILE,** , the compiled token may be of some different size and decompile, for example, as **SWAP OVER** , or may be non-decompileable.

A.OI.6.3.???? TOKEN> "token-from" OI-CODE
A.OI.6.3.???? TOKEN+ "token-plus" OI-CODE

The word **TOKEN+** is not necessarily equivalent to the phrase **1 TOKENS /XSWAP /+**. If the code memory address at the stack top points to a token compiled with **TOKEN,**, they are equivalent. But if the code memory address at the stack top points to a token compiled with **COMPILE,**, the word **TOKEN+** is allowed to add the size of that token instead of adding the default size of a token.

A.OI.6.4 The Open Interpreter threaded code access extension words
A.OI.6.5 The Open Interpreter in-line data access words

A.OI.6.5.???? /@ "slash-fetch" OI-INLINE
If return addresses are one-cell wide and code memory is data memory, and if alignment requirements for compiled tokens and data memory cells are different (that is, aligned code pointers are not aligned addresses), the system can implement only Class 3.

A.OI.6.5.???? /+ "slash-plus" OI-INLINE
A.OI.6.5.???? /ALLOT "slash-allot" OI-INLINE
A.OI.6.5.???? /GET "slash-get" OI-INLINE
A.OI.6.5.???? /PUT "slash-put" OI-INLINE

Code memory address units may have different size than data memory address units, and the phrase **1 CHARS /ALLOT 1 CHARS /ALLOT** may give different results than the phrase **2 CHARS /ALLOT**.

(The first phrase is guaranteed to reserve at least two code memory address units; on a Class 5 system, the second phrase may reserve only one code memory address unit, this happens if one code memory address unit is large enough to hold two characters.)

Since the words **/ALLOT** and **/+** may perform alignment on a code memory address unit boundary, the data elements in code memory must be accessed in the same way as they have been allocated.

A.OI.6.6 The Open Interpreter in-line data access extension words

These words are meaningful only on Class 4 and Class 5 systems. On a Class 5 system, unaligned code pointers cannot be placed onto the return stack, and these are the only words that can do something with an unaligned code pointer.

<end of document>

Forth and the Internet

FORML 21

The 21st annual FORML Conference was held November 21–23, 1999 (Friday through Sunday) at the Asilomar Conference Center. Asilomar is located west of Monterey, California, on a hillside that overlooks a wide sandy beach and the Pacific Ocean beyond. Most residence quarters and meeting rooms are clustered on the hillside. The cafeteria, chapel, auditorium, and administration building huddle at the base of the hill near the shore. The buildings have a natural, rustic design, and the pace at Asilomar is unhurried. The atmosphere is more that of a spiritual retreat, despite the number of conferences being held there.

The weather started off blustery, and rain began to fall by dinner time and on through the evening. By morning, however, the sky was clear and the surf was up. It was the same on Sunday when an outdoor barbecue was served for lunch, ending a picture-perfect day.

The Forth Interest Group was able to arrange for more comfortable rooms this year, especially for those of us who chose to double up as an economy measure. The meals were outstanding. Definitely a weekend to leave the diet behind.

The majority of those attending this FORML came from the United States, but we also welcomed two intrepid travelers from overseas. Federico de Ceballos came from Santander in northern Spain, and Charles Esson made the long flight from Ballarat, Victoria, in Australia.

FORML21 began at 3:30 p.m. with Richard Wagner presenting a tribute to Robert R. Reiling, who passed away a few months earlier. Mr. Reiling had guided FORML conferences for nearly 20 years, ever since they had been held at Asilomar. Like all those who work with quiet competence, his efforts will be appreciated most by those who must shepherd the conferences in the future.

The format of the conference then was outlined.

Though the theme of this year's FORML was "Forth and the Internet," the papers ranged over several topics:

Forth and Networking
Encryption
Text Processing
Forth Machines
Forth Philosophy
Teaching Forth

Charles Esson presented three papers on ColdForth, a Forth designed to run on Motorola's ColdFire family. This project was started two years ago after Motorola announced that the 68K family would be phased out. For fourteen years, Color Vision Systems, the company Charles is with, had been using the Motorola 68K family of chips, building a large base of Forth, assembler, and proprietary code. All of it was threatened with obsolescence. After exploring alternatives, they decided on ColdFire and Forth. Commercial Forths were looked at, but a critical requirement was preemptive multitasking, so they decided to roll their own.

To avoid a future problem with "disappearing hardware," ColdForth is being designed to be as portable as possible. To achieve this, the bulk of it is being written in ANS Forth. The compiler will do some optimizing to increase speed. Device drivers are coded as objects, so kernel words handling input/output or mass storage don't need to be rewritten—the appropriate driver is simply plugged in. The kernel code is being made GPLLable, and its source will eventually appear on the Internet. The hope is that others will adapt it to other chip families, and extend the list of device drivers.

Esson's first paper described the design of the TCP/IP protocol as a set of objects, just another device driver. In addition, an interface was developed to allow transfers on an ethernet link.

His second paper discussed the reasons for requiring a preemptive multitasker, and the design considerations to ensure its proper function.

The third paper described a heap to allocate and release memory for the datagrams processed by TCP/IP. The challenge was that it had to work without the benefit of an MMU, be written in Forth, and still be fast.

Federico de Ceballos dealt with the challenge of connecting a Forth program to a Siemens AS990 system. This is a distributed computer system having multiple nodes tied together by an ethernet. Each node is composed of a VE486 processor card, and a CP486 ethernet card. Each card has its own operating system (MICROS), and the two cards communicate via a MicroNET system using a TCP/IP stack! Federico presented the Forth code used to interface his program with the MicroNET C functions that handle data transfer. There was a high priority on reliability, as the program located the position of control rods in a nuclear reactor.

Skip Carter of Taygeta Scientific showed how to subset TCP/IP. TCP/IP is not a monolithic structure; instead, it is a collection of protocols. Not all are needed in a dedicated system, and careful pruning can simplify the code.

Wil Baden presented two papers on encryption.

The first was on the SHA-1 Secure Hash Algorithm. This is widely used to secure the Digital Signature Algorithm. His Forth code is designed to run on big-endian or little-endian Forths without change. Though it does much better on 32-bit systems, he has a 16-bit version available, including one for Quartus (Palm Pilot) Forth.

The second, "Solitaire," was about an encryption method used by a Soviet spy, and described by David Kahn in *Kahn on Codes*. The original method used a standard deck of 52 playing cards plus the two Jokers.The cards are arranged in Bridge

Continues on page 77.

Walter J. Rottenkolber • waltjr@sierratel.com
Mariposa, CA

Walter has "retired and drifted away from programming." He lives "out in the country near the edge of the universe" where he finds interest in the history of computers, Forth, and the implementation of the odd algorithm.

Linear Congruential Sequences

If you are like most of us, and need a few random numbers for a game or such, then a single *linear congruential sequence* (LCS) should be good enough. With a little more effort, we can get millions and millions of "random" numbers.

According to Knuth, (Chapter 3 of *The Art of Computer Programming)*, LCS was introduced by D.H. Lehmer in 1949.

Quoting Knuth,

We choose four magic integers,

```
m,     the modulus;           0 < m.

a,     the multipier;         0 <= a < m.

c,     the increment;         0 <= c < m.

X[0],  the starting value;    0 <= X[0] < m.
```

The desired sequence of random numbers (X[n]) is then obtained by setting

```
X[n+1] = (a*X[n] + c) mod m;
```

Knuth gives the following principles (paraphrased) for selecting those numbers.

(1) The "seed" number X[0] *may be chosen arbitrarily.*

(2) The number m *should be large, say at least* 2**30. *Conveniently it may be the computer's word size, since that makes the computation quite efficient.*

16-bit word size cannot satisfy this principle.

In Forth, we can write a LCS with *m* as the word size:

```
VARIABLE RAND-X

: RAND-NEXT  ( -- u )
   RAND-X @  a *  c +  DUP RAND-X ! ;
```

Another approach is to use for *m* an easily referenced large prime within the word size. For 32-bit arithmetic, 2**31–1, which is 2147483647, is a popular choice. The value of *c* should be 0.

```
: RAND-NEXT  ( -- u )
   RAND-X @  a m */MOD DROP  DUP RAND-X ! ;
```

(3) If m *is a power of 2, pick* a *so that* a mod 8 *is 5. If* m *is a power of 10, choose* a *so that* a mod 200 *is 21.*

This, with *c* as chosen below, ensures a cycle of *m* values that pass a test of "potency."

In *Starting Forth*, *a* is 31421.

(4) The multiplier a *should preferably be chosen between .01*m and .99*m, and its binary or decimal digits should* ***not*** *have a simple, regular pattern.*

31421 for a 32-bit word size fails this principle—it's too small.

Knuth recommends a "haphazard" constant like 3141592621. I call this "Pi21." It's enough digits of pi, with 21 tacked on, to fill a 32-bit word. I think *Starting Forth*'s 31421 was chosen the same way for 16-bit words.

I remember Pi21 by "Now I want a large container of coffee-21."

(5) The value of c *is immaterial when* a *is a good multiplier, except that* c *must have no factor in common with* m.

So 1 or *a* look like good values for *c*.

This gives:

```
: RAND-NEXT ( -- u )
   RAND-X @  3141562621 *  1+  DUP RAND-X ! ;
```

or

```
: RAND-NEXT ( -- u )
   RAND-X @  1+  3141562621 *  DUP RAND-X ! ;
```

(6) When m *is the word size the least significant digits of random numbers are not very random, so decisions based on the number should always be influenced primarily by the most significant digits.*

In other words, don't use **MOD** to select a value. The *Starting Forth* function is good when *m* is the word size.

```
: RAND-UNIF ( u -- n ) RAND-NEXT UM* NIP ;
```

Starting Forth calls this **CHOOSE**.

(7) The randomness in t *dimensions is only one part in the* t*-th root of m.*

Don't use a LCS for simulations requiring high resolution.

(8) At most m/1000 *numbers should be generated; otherwise the future will behave more and more like the past.*

For a 32-bit word size, a new scheme or a new multiplier should chosen every few million random numbers.

For a 16-bit word size, a new scheme or a new multiplier should chosen every few 64 or 65 random numbers.

From this, you can see that you can't get a good single LCS for 16-bit arithmetic. This can be fixed by using multiple LCSs or other methods. Later we'll give a multiple LCS definition for 16-bit Forth.

Wil Baden • Costa Mesa, California
wilbaden@netcom.com

Wil Baden after many years of profane language has retired to Standard Forth. For a copy of the source for this article, send e-mail requesting Stretching Forth #26: Linear Congruential Sequences.

Exhibits

Here is an exhibition of LCSs that have been popular. I have assigned names for ease of reference here.

Definitions that do not return 0 should not be initialized with 0.

Some of the tests that have been made on the sequences are named, but details are postponed to another article.

PI-RAND is LCS with the multiplier based on PI. This is given in the summary of LCSs in the last section of chapter 3 of Knuth's *The Art of Computer Programming*, all editions.

```
: PI-RAND-NEXT  ( -- 0..4294967295 )
   RAND-X @
      3141592621 *  1+
   DUP RAND-X ! ;
```

SJ-RAND was proposed by Lewis, Goodman, and Miller in the *IBM Systems Journal* in 1969. It was used in APL and IMLS subroutine library. It was also an option in SwiftForth[1]. The main reason for continued use is that the square of *a* is less than modulus *m* and it can be implemented without arithmetic overflow. However, such small multipliers have known defects. (16807 is 7**5.)

```
: SJ-RAND-NEXT  ( -- 1..2147483646 )
   RAND-X @
      16807 2147483647 */MOD DROP
   DUP RAND-NEXT ! ;
```

EASY-RAND was nominated by George Marsaglia (1972) as a candidate for the best multiplier, perhaps because 69069 is easy to remember.

```
: EASY-RAND-NEXT  ( -- 0..4294967295 )
   RAND-X @
      69069 * 1+
   DUP RAND-NEXT ! ;
```

BS-RAND uses the best spectral primitive root for modulus 2147483647. G.S. Fishman found it by brute force in 1986.

```
: BS-RAND-NEXT  ( -- 1..2147483646 )
   RAND-X @
      62089911 2147483647 */MOD DROP
   DUP RAND-X ! ;
```

EP-RAND is an efficiently portable multiplier found by Fishman in 1988.

```
: EP-RAND-NEXT  ( -- 1..2147483646 )
   RAND-X @
      48271 2147483647 */MOD DROP
   DUP RAND-X ! ;
```

EP2-RAND is an efficiently portable multiplier found by L'Ecuyer in 1988.

```
: EP-RAND-NEXT  ( -- 1..2147483338 )
   RAND-X @
      40692 2147483647 248 - */MOD DROP
   DUP RAND-X ! ;
```

SF-RAND is the random-number generator from Brodie's *Starting Forth*. Of course, with 16-bit arithmetic, **65535 AND** may be omitted.

```
: SF-RAND-NEXT  ( -- 0..32767 )
   RAND-X @
      31421 *  6927 +  65535 AND
   DUP RAND-NEXT ! ;
```

C-RAND is the default random-number generator for the Standard C Library.

C-RAND-NEXT is C's **RAND** .

```
: C-RAND-NEXT  ( -- 0..32767 )
   RAND-X @
      1103515245 *  12345 +
   DUP RAND-X !
   16 RSHIFT  32767 AND ;
```

RANDU is the egregious RANDU of the '60s and '70s. It must be initialized to odd values only. Note that the multiplier in hex is 10003. For any three successive values, $9X - 6Y + Z$ is **0 mod 2147487648**.

```
: RANDU-NEXT  ( -- 1..2147483647 )
   RAND-X @
      65539 *  2147483647 AND
   DUP RAND-X ! ;
```

Efficiently Portable Implementations

In the *m*-is-word-size definitions, the *n* low-order bits cycle in a 2**n period, as mentioned in (6) above.

```
( n is 1 )
:GO CR 17 0 DO  PI-RAND-NEXT 1 AND . LOOP ; GO
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

( n is 2 )
:GO CR 17 0 DO  PI-RAND-NEXT 3 AND . LOOP ; GO
2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2

( n is 3 )
:GO CR 17 0 DO  PI-RAND-NEXT 7 AND . LOOP ; GO
7 4 5 2 3 0 1 6 7 4 5 2 3 0 1 6 7
```

Etcetera.

In *m*-is-word-size definitions, the second half of the period is the same as the first half with sign-bit inverted. The quarter-cyles are also the same except for the two top bits.

Thanks to the first edition of *The Art of Computer Programming*, I've used the function PI-RAND (with various names) in different languages for 30 years. But things are bigger and faster than they used to be. Typically, I now use random numbers to test something a million times. So I suggest for a simple, powerful, random-number generator for today, to combine

EP-RAND and EP2-RAND, as proposed by Knuth.

```
    65536 0= NOT [ IF]

  1 VARIABLE RAND-X  VARIABLE RAND-Y

  3 : RAND-NEXT  ( -- 1..2147483647 )
  4    RAND-X @
  5       48271  2147483647  */MOD DROP
  6       DUP RAND-X !
  7    RAND-Y @
  8       40692  2147483399  */MOD DROP
  9       DUP RAND-Y !
 10    -  DUP 0> NOT IF  2147483647 +  THEN ;

    [ THEN]
```

According to Knuth, the period is about 74 quadrillion.

RAND-X and **RAND-Y** should be initialized independently to non-zero values for best results.

RAND-X does not have the problem with low-order bits.

```
 1 RAND-X !  1 RAND-Y !

 ( n is 1 )
 :GO CR 17 0 DO  RAND-NEXT 1 AND . LOOP ; GO
 1 1 0 1 0 1 1 1 0 1 1 0 0 0 1 1 0

 ( n is 2 )
 :GO CR 17 0 DO  RAND-NEXT 3 AND . LOOP ; GO
 3 2 0 3 3 3 0 3 0 0 2 1 2 0 3 2 2

 ( n is 3 )
 :GO CR 17 0 DO  RAND-NEXT 7 AND . LOOP ; GO
 5 7 5 3 5 6 0 3 5 2 5 5 3 4 3 2 4
```

With 16-bit arithmetic Forth, *three* LCSs should be combined. The following is adapted from a suggestion by L'Ecuyer in 1988.

```
    65536 0= [ IF]

  1 VARIABLE RAND-X
  2 VARIABLE RAND-Y
  3 VARIABLE RAND-Z

  5 : RAND-NEXT  ( -- 1..32363 )
  6    RAND-X @  157  32363  */MOD DROP
  7       DUP RAND-X !
  8    RAND-Y @  146  31727  */MOD DROP
  9       DUP RAND-Y !
 10    -  DUP 0> NOT IF  32363 +  THEN
 11    RAND-Z @  142  31657  */MOD DROP
 12       DUP RAND-Z !
 13    -  DUP 0> NOT IF  32363 +  THEN ;

    [ THEN]
```

The exhibits here were originally presented in profane languages—outside of the temple of Forth. Some of those languages do not have a double accumulator for multiplication, and may fail with integer overflow. When *a* times *a* is less than *m*, and the remainder *m* divided by *a* is less than the quotient, then **a m */MOD DROP** can be replaced by:

```
 [  m a / ] LITERAL /MOD
 [  m a MOD ] LITERAL * >R
 a *  R> -
 DUP 0< IF  m +  THEN
```

All intermediate results will be in the official range of integers.

Such substitutions can be made in the 32-bit and 16-bit versions of **RAND-NEXT**. The practice will be more useful in the reverse direction, converting profane language to Forth. Many conversions from Fortran or another profane language to Forth use a precomputed quotient and remainder. For a Forth definition, ***/MOD DROP** can be used to simplify.

Don't Use MOD

You have been warned not to use **MOD** to select random numbers. With a small range, the problem is neglible, but the following shows the danger.

```
    65536 0= NOT [ IF]

  1 1 RAND-X !  1 RAND-Y !

  3 CREATE BINS  10 CELLS ALLOT

  5 :GO CR CR ." Distribution Using MOD " CR
  6    BINS 10 CELLS ERASE
  7    20000 0 DO
  8       \  9 Digit Number
  9       RAND-NEXT 1000 MIL MOD   ( u)
 10       \  Into 10 BINS
 11       100 MIL /
 12       CELLS BINS +  1 SWAP +!  ( )
 13    LOOP ; GO

 15 :GO
 16    10 0 DO
 17       CR  I CELLS BINS + @
 18       DUP 4 .R SPACE
 19       50 / [ CHAR] * EMITS
 20    LOOP ; GO
```

Distribution using MOD:

```
2805 ******************************************************
2288 *********************************************
1816 ***********************************
1871 ************************************
1828 ***********************************
1807 ***********************************
1908 ************************************
1940 ************************************
1884 ***********************************
1853 ***********************************
```

```
22 : RAND-UNIF ( +n -- u ) 2* RAND-NEXT UM* NIP ;

24 :GO CR CR ." Distribution Using Multiply " CR
25    BINS 10 CELLS ERASE
26    20000 0 DO
27       \  9 Digit Number
28       1000 MIL RAND-UNIF       ( u)
29       \  Into 10 BINS
30       100 MIL /
31       CELLS BINS +  1 SWAP +!  ( )
32    LOOP ;  GO

34 :GO
35    10 0 DO
36       CR  I CELLS BINS + @
37       DUP 4 .R SPACE
38       50 / [ CHAR] * EMITS
39    LOOP ; GO
```

Distribution using multiply:

```
1977 ************************************
2042 ************************************
1974 ************************************
2024 ************************************
2021 ************************************
2007 ************************************
1981 ************************************
1991 ************************************
2001 ************************************
1982 ************************************

    [ THEN]
```


Tools for Linear Congruential Sequences

Note: The preceding "Tool Belt" article was #9. From now on, the "Tool Belt" article number will agree with the current "Stretching Forth" article number. Using two sets of numbers confuses me.

Definitions for the non-standard words in the current "Stretching Forth" article are given here for your convenience. Most of these have appeared in previous articles, and will have little or no discussion.

```
 1 : MACRO  ( "name <char> ccc<char>" -- )
 2    :   CHAR PARSE POSTPONE SLITERAL
 3    POSTPONE EVALUATE
 4    POSTPONE ; IMMEDIATE ;

 6 MACRO :GO  " ANEW NONCE  : (GO) "
 7 MACRO GO   " (GO) NONCE "

 9 : POSSIBLY  ( "word" -- )
10    BL WORD FIND   ( xt flag)
11    ?DUP AND IF  EXECUTE  THEN ;

13 : ANEW  ( "name" -- )
14    >IN @ POSSIBLY  >IN ! MARKER ;

16 \  NOT  is equivalent to `0=` or `INVERT`.
17 MACRO NOT  " 0= "

19 \  MIL  is convenient for large numbers.
20 : MIL  ( n -- 1000000**n )  1000000 * ;

22 \  EMITS  displays a CHAR n times.
23 : EMITS  ( n CHAR -- )
24    SWAP  0 ?DO  DUP EMIT  LOOP DROP ;
```

Wil Baden • Costa Mesa, California
wilbaden@netcom.com

Wil Baden after many years of profane language has retired to Standard Forth. For a copy of the source for this article, send e-mail requesting Standard Forth Tool Belt #26: Tools for Linear Congruential Sequences.

HC11 Code Interface to

Dallas Semiconductor Information Buttons (iBs)

The Dallas Semiconductor Information Button, or iB, is a non-volatile RAM, with a serial interface, carried in a two-contact housing, often using a button-battery style enclosure. The name of these devices has itself undergone a sort of evolution at Dallas; originally they were called "Touch Memories," then they were known as "Information Buttons," and then as "AutoIdentification" devices. Of all this, we at Jarrah Computers have stuck with the "Information Button" nomenclature. There is an entire family of devices in the iB range, all denominated DS19XX. The family include memory of various sizes, one contains a timer, and one a thermometer. Jarrah Computers have primarily used the DS1996 (64Kbit NV RAM), and this article is based on code which interfaced to the 1996, but we believe the processes are similar for all members of the family.

Dallas have promoted the iBs as "attaching digital data to physical objects"—their intended applications being in industry to track, say, gas cylinders, shipping containers, and the like. To this end, there is quite a bit of information in the Dallas data sheets (available on-line at www.ibutton.com) on how to run connections to the iB wherever it is located on its "physical object"—sometimes a couple of metres of wire are used; some hookups use metal straps, or a set of cables, and connectors to wire the iB from wherever it is physically located back to a reading/writing device.

This is worth noting, as any interface code must almost presume that the contacts to the iB may be momentarily high resistance—that contacts may be making and breaking as we try to communicate. The retries in the code are primarily to handle momentary breaks in connection. The example given at the end of this article, on backup memories, is an example of attaching digital data (in iB) to a CPU board, which is, in the end, a physical object!

The memory in the iB is organised into a set of 32-byte pages, plus a 32-byte scratchpad, via which all reads and writes are mediated. Furthermore, Dallas recommend that the data be formatted into what they call Universal Data Packets, which is a count byte, the data, then a two-byte CRC, meaning that only 29 bytes of actual data can be stored in each 32-byte page. The UDP, including checksum, provides an extra check that good data has been read and is, in our view, well worth the extra effort.

There are two modes for dealing with iBs, called *regular* and *overdrive* speeds by Dallas—we will only be dealing with the regular speed, as overdrive requires processors which are faster than a (2 MHz) 68HC11—they typically deal in 2-10µS periods, which is faster than some 68HC11 instructions!

The bi-directional serial interface is implemented on a single active wire, and ground. The recommended processor interface is a single bi-directional port line (and ground), with the port line tied to Vcc (+5V) via a (nominal) 4K7 resistor. The iB sends data by simply generating current pull when it wants to send a zero, and the current pulls the resistor voltage down far enough to be sensed as a low on the (input) port line. To send a "one" the iB, of course, does not need to do anything.

The serial protocol involves commands being sent out to the iB, and the iB sends the required data back over the same wire, so the processor must switch the port line to input mode before the iB starts sending the data. As reading data (and thus switching the direction of the port line) is involved in all interactions between a processor and the iB—and as we must ensure that we are never driving the line when the iB is—synchronous dynamic control of the bi-directional port line's *direction* is a critical part of the interface to the Dallas iBs.

The most important aspect of the serial protocol coding is timing. The processor must drive and sense the line during the allocated time slots, so the time-critical routines must be worked out almost on a cycle-by-cycle basis.

1. iB Interface Details

There are four basic processes in dealing with the iBs:

1.1 Reading "presence"—refer to Figure One

To read iB presence, the port line (in output mode) must be driven low for greater than 480 µS, and then the port line must be sampled (in input mode), starting between 15 and 60 µS after the low pulse is released, and the iB will drive the line for between 60 and 240 µS. If an iB is present, there will be a "low" generated by the iB; if no iB is present, the line will of course stay high due to the tie-high resistor.

1.2. Reading a data bit—refer to Figure Two

To read an iB data bit, the port line (in output mode) must be driven low for between 1 and 15 µS (basically, as short as possible!), then the port line must be sampled (in input mode), starting as soon as the low pulse is released, and the iB will drive the line for a window of 15 µS. If the iB data is zero, there will be a "low" generated by the iB; if the iB data is one, the line is left high.

1.3. Writing a "one" bit—refer to Figure Three

To write an iB data bit "one," the port line (in output mode) must be driven low for between 1 and 15 µS (basically, as short as possible!), then the port line is left high for the remainder of the 60 µS window.

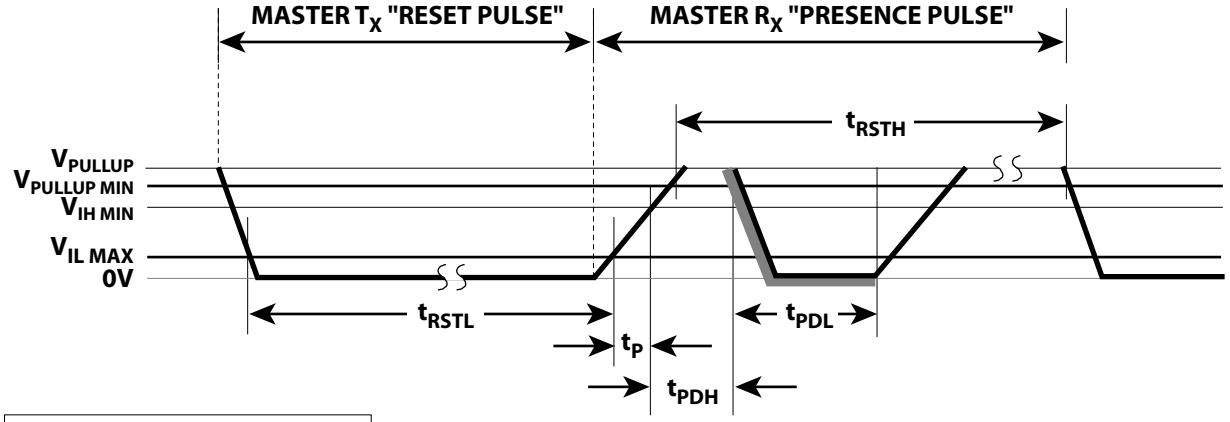
1.4. Writing a "zero" bit—refer to Figure Four

To write an iB data bit "zero," the port line (in output mode) must be driven low for the entire 60 µS window.

Dave Edwards • jarrah@inf.net.au
Perth, W.A., Australia

Dave Edwards formed Jarrah Computers, an embedded systems development company using primarily Forth, in 1984. His company has specialised in design of custom microcontrollers. He is writing his second opera, using his own Forth-programmed MIDI music system.

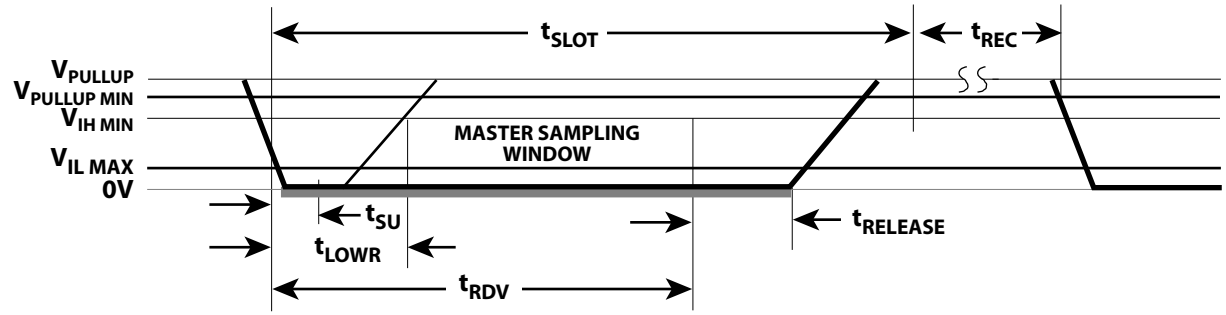
Figure One. Initialization procedure "reset and presence pulses"



Regular Speed
480 µs ≤ t_{RSTL} < ∞
480 µs ≤ t_{RSTH} < ∞ (includes recovery time)
15 µs ≤ $_{PDH}$ < 60 µs
60 µs ≤ t_{PDL} < 240 µs

Overdrive Speed
48 µs ≤ t_{RSTL} < 80 µs
48 µs ≤ t_{RSTH} < ∞
2 µs ≤ t_{PDH} < 6 µs
8 µs ≤ t_{PDL} < 24 µs

Figure Two. Read-data time slot



Regular Speed
60 µs ≤ t_{SLOT} < 120 µs
1 µs ≤ t_{LOWR} < 15 µs
0 ≤ $t_{RELEASE}$ < 45 µs
1 µs ≤ t_{REC} < ∞
t_{RDV} = 15 µs
t_{SU} < 1 µs

Overdrive Speed
6 µs ≤ t_{SLOT} < 16 µs
1 µs ≤ t_{LOWR} < 2 µs
0 µs ≤ $t_{RELEASE}$ < 4 µs
1 µs ≤ t_{REC} < ∞
t_{RDV} = 2 µs
t_{SU} < 1 µs

All figures re-drawn from diagrams provided courtesy of Dallas Semiconductor.

2. The Code

Having described the basic processes, now we can look at the code which does all of this (refer to the listings). This is where some detailed knowledge of the 68HC11's opcodes and opcode execution times is needed; this is where we start counting cycles!

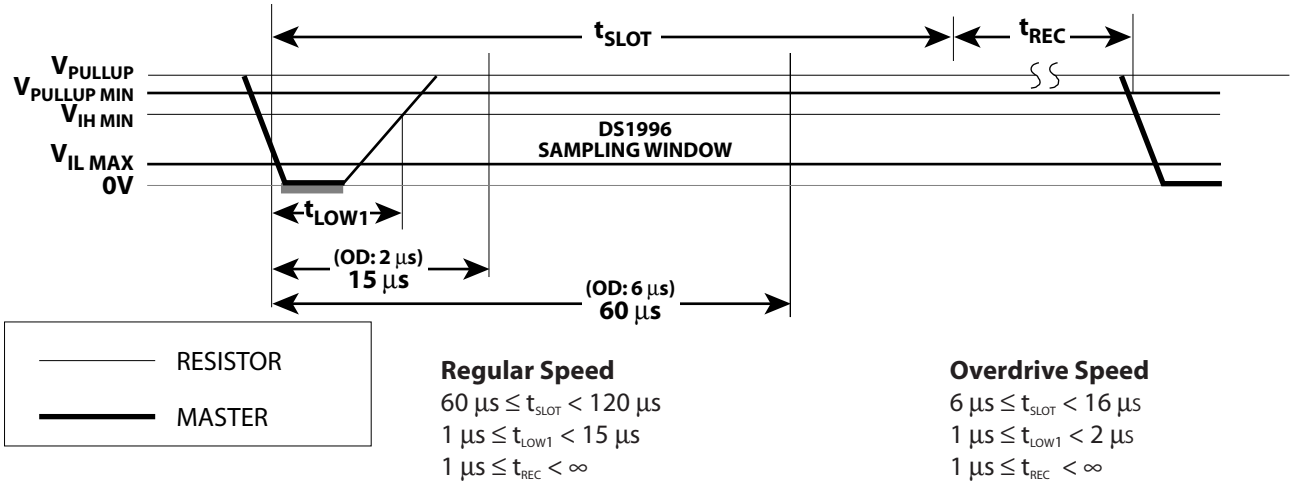
2.1 Coding Preliminaries

Before I begin the detailed discussions of the coding, as Jarrah Computers are a Forth-based company, we have developed our own 68HC11 assembler and therefore must provide some background to our style of coding, just to help you understand the listing.

The assembler uses *reverse Polish assembler*, so what would normally be written as, say, `LDD # $1234` (load'D' immediate with $1234) is written `(HEX) 1234 # LDD`, and the opcode names all end with `,` to indicate the point in the code where assembly actually occurs. We can also say `VARIABLE-NAME LDD` and the like.

The stack referencing used in this code is similar to that of many Forths: the Y register points to the TopOfStack (High byte), so we use `,Y` addressing mode: `0 ,Y` references TOS, `2 ,Y` references NOS, and so on. We also have our own definitions for RAM locations: RAM arrays are called `RARRAY`, used `n RARRAY Name`. `VAR`s are our `VARIABLE`s (a "`2`" `RARRAY`), and `CVAR`s are single-byte variables. The HC11 has two accu-

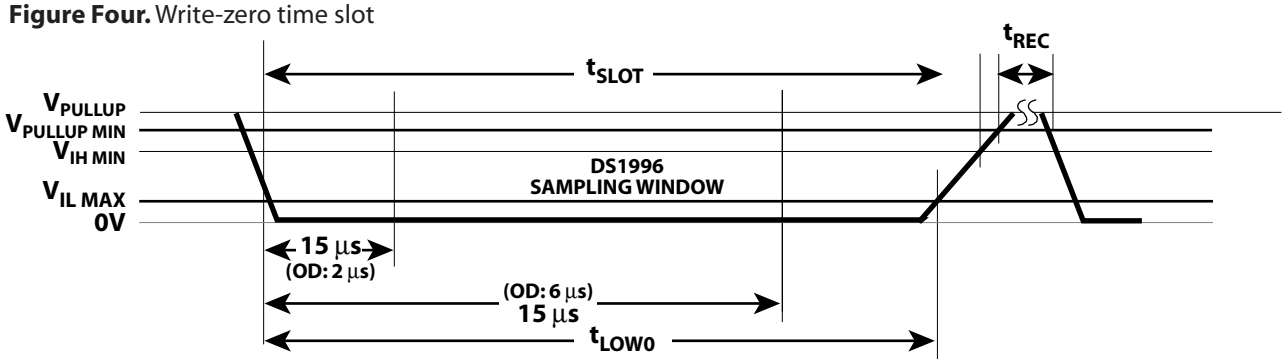
Figure Three. Write-one time slot



Regular Speed
60 µs ≤ t_{SLOT} < 120 µs
1 µs ≤ t_{LOW1} < 15 µs
1 µs ≤ t_{REC} < ∞

Overdrive Speed
6 µs ≤ t_{SLOT} < 16 µs
1 µs ≤ t_{LOW1} < 2 µs
1 µs ≤ t_{REC} < ∞

Figure Four. Write-zero time slot



Regular Speed
60 µs ≤ t_{LOW0} < t_{SLOT} < 120 µs
1 µs ≤ t_{REC} < ∞

Overdrive Speed
6 µs ≤ t_{LOW0} < t_{SLOT} < 16 µs
1 µs ≤ t_{REC} < ∞

mulators, A and B, and these are indicated by using `.A` or `.B` respectively.

Words are defined by the bracketing `HC:` and `;HC` defining words, which basically perform the same functions as `:` and `;` in Forth, or `CODE` and `END-CODE` in Forth assemblers. The assembly address of a `HC:` *name* is left on the stack when the *name* is invoked. For subroutines, we have also developed the defining words `SUB:` and `;SUB` which have the following characteristics:

`;SUB` compiles an RTS, instruction before doing the "end-definition" work of `;HC`, and `SUB:` makes a header similar to `HC:` with the additional `DOES>` property of compiling a JSR to the *name*'s address.

This means that a word defined by `SUB:` can be invoked simply by its name. I.e., it does not need to be invoked by saying `NAME JSR`, which is what `HC:` *name*s require. This has the net effect of making the code look very much like high-level Forth. For example, if we have `SUB:` routines like:

```
SUB: OVER ( code instructions ) ;SUB
SUB: SWAP ( code instructions ) ;SUB
SUB: +    ( code instructions ) ;SUB
```

...we can simply write, in a subsequent definition:

```
SUB: NEW-WORD     SWAP OVER + (etc.)    ;SUB
```

In fact, the code becomes almost simultaneous Forth *and* assembler, in that it's possible to have a set of statements that look like Forth, and then have opcodes one after another as in assembler. For example, at the end of the word `MAKE-UDP`, we have:

```
..... CRC16 LDB, FF # .B EOR,
    iBCRC LDA, FF # .A EOR, PSHD
    SWAP ! "UDPBUFF" SWAP OVER C@ "3" + ;SUB
```

The first two lines are almost pure assembly, but the last is a lot like Forth.

2.2 The iButton Code—The Bits

With those preliminaries out of the way, we can now proceed to examine the code in detail. After the Forth kernel and the two CRC lookup tables, the iB code proper starts. First of all, we name the port line (an address, and a bit mask) and its associated Direction Register line (again, an address

and a bit mask). The port line is kept normally in input mode to ensure that we never drive the line when the iB is attempting to drive it.

We start with some timing functionals which are used to build the "bit interface" words. The first word is a utility timer, `tmwait`, which expects a number in the B accumulator ("*n*"), and consumes 5*n*+11 cycles—the 11 being the call/return opcodes. We then use a utility compiler word, `uSwait`, which takes a given number on the assemble-time stack (representing microseconds) and converts it into a number which, when fed into `tmwait` via the B accumulator, will consume that number of microseconds. For example, the phrase:

```
5 uSwait # LDB, tmwait
```

compiles code to wait for five microseconds.

The same values work in the word `WRLO` ("write low"), which takes a number on the stack (the machine's run-time stack!) and writes a low on the iB port line for that number of microseconds. Thus, the phrase:

```
5 uSwait # CPUSH WRLO
```

compiles code to write a low on the iB line for five µsecs.

2.3 Reading "Presence"

This process tests the iB line to see if a device is present on the line—and this presence-testing is also referred to as "reset"ing the device, so our first word is called `readiBreset (RDiBRST)`, and returns its result in a condition code register (0= flag). How `RDiBRST` works is:

We set interrupt masks (to lock out any interrupts and thus guarantee the timing), and then write a low pulse for approximately 450 µsecs—the iB interprets such a long pulse as a request for a presence pulse response, which it will now send.

We enter a timed loop to consume 480 µsec—the execution time of the loop was calculated (33 cycles), and then the count (29) was determined so that 480 µsec was consumed (29*33 = 957 cycles = 479 µsec). During this loop, we monitor the iB line to see if it goes low at any time; if it does, we have an iB present. Note that, if we sense the low, we do not immediately exit, but remain in the loop for the full 480 µsec.

`RDiBRST` is used primarily to reset communications; it is recommended that the presence checking be run as the start of any communication sequence.

The word `iBPRESENT` simply runs `RDiBRST` and exits in Error! mode if the device is not found. The details of this error handling are implementation dependent, but the general idea of exiting to the system level on any iB errors is to simplify control structures later in the code, and is highly recommended.

3. The Bytes

3.1 Sending a Byte to the iB

The first byte-level interface word is `!iBYTE`, which sends a byte to the iButton. A one is loaded and stored in utility register N—this is a mask (rotated left once for each bit) against which the input byte is tested to determine the state of each bit. It is also used as the default counter, as this bit-rotating out of the accumulator into the carry register is taken as the termination test.

The process then tests each bit of the input byte, and if the bit is high, the "transmit one" process is undertaken, otherwise "transmit zero" is undertaken. Note that these processes are guarded by disabling interrupts (setting of the interrupt mask) before the processes start, and re-enabling interrupts after the 60 µsec window, whichever path is taken, again to guarantee timing.

3.1.1 "transmit one" (refer to 1.3) takes the line low for 3 µsec, and then leaves the line high for a remaining 57 µsec. The 3 µsec low pulse is generated as follows:

a. Write a low to the port line—remember that at this point the port is still in INPUT mode, therefore the tie-high resistor will be holding the line high.
b. Make the port line output—this is when the port's output driver comes into play, and drives the line low. Our timing of the low pulse out starts at this point.
c Make the port line input—this is where the tie-high resistor takes over again, so the port releases the line, and the resistor pulls the line high, only 3 uSec after it was taken low.

3.1.2. "transmit zero" (refer to 1.4) takes the line low for the entire 60 µsec window, so the iB is guaranteed of reading a low bit.

3.2 Reading a byte From the iB

We now examine the word which reads a byte from the iB, `@iBYTE`. Again, this is an eight-step loop, reading a bit at a time. Room is made on the stack for the result, eight is stored as a count, and we begin:

- Write a low pulse for as short a time as possible. (again, a 3 µsec low)
- Wait 5 µsec before commencing sensing:
- Monitor the iBline for next 100 µsec:
  If it is pulled low, accumulate a "zero" bit
  If it is *not* pulled low, accumulate a "one" bit and repeat this process for the eight bits.

This completes the fundamental interface to the iBs. We can now get bytes into and out of them. It is worth noting that, while all of the processes up to now have involved critical timing, other words can call them completely asynchronously—the timing is wrapped up within these words!

4. The Communications—Strings of Bytes

Now we can start the "communications" level of talking to iBs. As we pointed out before, all interactions with the iB is via a buffer page called the scratchpad. Data must be written to the scratchpad (indicating the final target address at this point), verified *in* the scratchpad (which involves reading all of the data back and comparing it to the source), and finally copying the data in the scratchpad into the final target page.

Dallas provide a flowchart as a guide to this layer. There are sets of functions which the iB can perform—`READ ROM`, `MATCH ROM`, `SEARCH ROM`, and `SKIP ROM`—and all the general memory functions come under the `SKIP ROM` sub-section. This is why we have `iBSKIP`, a "start communications" word which checks for iB presence and then sends the "skip" command ($CC). The `SKIP-ROM` commands used are `READ-ROM`, `WRITE-Scratchpad`, `READ-Scratchpad`, and `COPY-Scratchpad`. `READ-ROM` is used for general reading of data (page-based, ran-

dom page access), and the others are used as a set to write data to a target page in memory via the scratchpad.

4.1 Moving Data to the Scratchpad—SXMOVE

We start with the word which moves data to the scratchpad (which has been shortened to SX in word names), namely SXMOVE, which uses control data on the stack—the address in the CPU's memory in which the data is stored, the target iB address, and finally the count (which *must* be less than or equal to 32 bytes). We do iBSKIP, then send the command byte $0F ("Write Scratchpad") followed by the two bytes of the target address (the 16-bit address split into two separate bytes) and the count, then the actual data is sent byte-by-byte. Note that all three numbers are left on the stack for subsequent words to check data against, and that the iB does not need to know the source address.

4.2 Checking the Scratchpad—SXOK and SXCHK

Next is the code to check the status of the move. Firstly, we use RDSX to get status information from the iB. After iBSKIP, RDSX sends the command byte $AA ("Read Scratchpad"), then reads back the next three bytes from the iB; these bytes represent the target address (as two separate bytes on the stack, just as we sent them), and the E/S status register.

The main scratchpad-checking word is SXCHK. RDSX is run to get the SXMOVE result data, then the two target address words are checked against the target address still on the stack from sending the data, and then the E/S status byte is checked; if any of the uppermost three bits of E/S are set, there has been an error. Hence we compare to $20—if the number is between $00 and $1F, it is okay. If any errors are encountered during these checks, the carry flag is set (in which case we do iBRST to clear the iB's communication and leave a set carry to indicate the error), and we proceed only in the case of a clear carry.

If things are all okay, we proceed to check every byte read back from the scratchpad against the bytes in memory from whence they were moved (using the source address on the very bottom of the stack). Again, any error causes carry to be set, which this time is left as the return code from the entire word.

We check the E/S byte in the word SXOK—similar to SXCHK, if any of the uppermost three bits of E/S are set, there has been an error, so we again compare E/S to $20 (after dropping the address bytes).

4.3 Copying the scratchpad—SXCOPY

The last word of this section is the scratchpad copy command, which gives the iB the all-clear to write what is currently *in* the scratchpad to the target address. After the iBSKIP we send the command byte $55 ("Copy Scratchpad"), and copies of the target address (again in separate bytes). Finally, we can drop the six numbers we have accumulated on the stack!

4.4 The Universal Data Packet (UDP).

To complete this section, we will look at the Universal Data Packet format recommended by Dallas Seminconductor. The UDP format is a count byte at the start, the data and then a two-byte checksum (CRC). This means we can get a maximum of 29 net bytes in a 32-byte memory page. The checksum is calculated from lookup tables provided by Dallas Semiconductor, as is the algorithm to perform the computation on bytes. There are two lookup tables, one for the upper part of the CRC, and one for the lower. The algorithm for computation is:

Input byte XORed with lower CRC, and the result of this is used as an index into CRC16HI to fetch the new high byte of the CRC. The result of the XOR is then used as the index into CRC16LO, and the fetched byte is XORed with the previous value of the higher CRC and the result is the new value of the lower byte of the CRC. At the end of the process, we invert all the bits by XORing with -1. This algorithm is coded in the word UDPCRC.

Using UDPCRC, we develop the word MAKE-UDP, which takes a source address, iB target address (which is actually not used *in* MAKE-UDP), and the count. The outputs are a new source address (the actual UDPBUFFer), the unaltered iB target address, and the original count incremented by 3. We have created a buffer UDPBUFF which is 32 bytes long as a staging area for this process. Firstly, we store the original count at UDPBUFF, and then CMOVE the bytes from the input address to UDPBUFFer+1. The last bit of the process is to calculate the CRC, by clearing our CRC accumulation variables, and then going through the source data and running UDPCRC on each fetched byte. The result is stored in the UDPBUFF at the end of the data.

5. iBMOVE and iBMOVE>.

So we can now approach the "final" words, which we have named iBMOVE (to move data *to* the iB) and iBMOVE> (to move data back *from* the iB to the CPU's memory). To start with, we develop iBMOVE? and iBMOVE>?, which attempt a read/write and leave the carry flag as the result—a set carry (CS) means the attempt was okay. If these words fail, the main words (iBMOVE and iBMOVE>) administer a set of retries.

iBMOVE? uses a source address, the iB target address, and a count. iBMOVE? runs SXMOVE and then SXCHK. If all is okay, SXCOPY is run, with a subsequent SXOK to check that the move itself generated no errors. If all is *not* ok, the arguments are dropped, and a clear carry is left to indicate a fault.

iBMOVE>? again uses a destination address, the iB source address, and a count. iBMOVE>? seeds the CRC variables with the iB *page* address, which is calculated by dividing the actual address by 32. We then run iBSKIP followed by the command byte $F0 ("Read Memory"). We then send the target address (again as two separate bytes), and then a loop of reading the data from the iB, and we once again use the UDPBUFF as in interstitial area to read the data—and we calculate the CRC using UDPCRC as we read each byte.

Seeding the CRC variables with the iB page address guarantees that the final result is $B001 (after CRCing the CRC in the iBpage itself!), so this check is run to leave the final result in the carry flag.

Finally, iBMOVE uses the same arguments as iBMOVE?, and iBMOVE> uses the same arguments as iBMOVE>?.

With iBMOVE, we first make the input data into a UDP, set up a counter to count retries (Jarrah Computers use 20), and then begin with trials of iBMOVE?. If iBMOVE? is successful, we are done; but if it is not, we wait for about 2 µsecs (to allow for iB recovery, in the case of a fault), decrement our counter, and try again. If we do not get a good write in our *retries* number of attempts, we must indicate an Error! to the system.

With iBMOVE>, we first increment the count by three (we know the iB UDP will be that much larger than the data bytes

we have been asked for), and the UDPBUFF is set up as the address to read the data into. We again set up a retry counter to 20, and begin attempts to read the data with iBMOVE>?. If errors are indicated, we decrement our counter and try again (and again, if we exhaust the retries we must indicate an Error! to the system). If the move from the iB was successful, we then CMOVE the data from the UDPBUFF to the final destination address. These words are used by simply writing:

```
Address iBAddress  Number  iBMOVE
```
to write data to the iB

```
Address iBAddress  Number  iBMOVE>
```
to read data from the iB

6. Example Application—iBak, iBVars, iBArrays…

As an example of an application using the above interface, Jarrah Computers have used an iB as a backup memory for data used by a control system. The only difficulty of implementing this is that we now need two different references to the same variable (array, etc.)—its address in RAM and its address (page-based) in the iB, which we shall call the iBak in this application. To overcome this, a few simple Forth words solve the problem.

To begin with, we make a note of the starting RAM address of the backed-up variables (iBAKram), and declare the starting address of the iBak (iBAKstart). Also, when we have reached the end of declaring the backed-up variables, we can calculate the total size of the area used (iBAKsize). We will be storing 29-byte (29 is called iBDATA) chunks of RAM into iB pages of 32 bytes (32 is called iBPAGE). This means that, to back up even a single byte, the entire 29-byte page will be written from memory into the iB. The following table summarises the correlation between RAM addresses and iB adresses.

RAM Adresses		**iB Page**
BASE Address	*Range*	
iBAKram	..(to+28)	iBAKstart
iBAKram+ iBDATA	..(to+28)	iBAKstart+ iBPAGE
iBAKram+2* iBDATA	..(to+28)	iBAKstart+2* iBPAGE
… etc.		

Note that the RAM addresses are proceeding in steps of 29 (iBDATA), while the iB addresses are proceeding in steps of 32 (iBPAGE). This complexity is part of what we are trying to hide here! The RAM addresses in the left-hand column represent base addresses which correspond to pages in the iB—any RAM address between a base address and the base address+28 will use the base address, and a count of 29, in reads from, and writes to, the iBak.

To implement our interface, we develop some utility words:

A>PG takes a RAM address and returns a base RAM address which corresponds to the start of the associated iB page.

Ac>PGS takes a RAM address and count, and returns the base of the starting page, the base of the ending page, and a count of the number of iB pages consumed by the original count. As the interface to the iB is via arrays (i.e., address count), the job of this word is to work out whether the RAM array crosses any iB page boundaries, and thus requires multiple page accesses.

For example, if a two-byte variable occupies the last byte of one iB page and the first byte of the next iB page, Ac>PGS will return the base address of the first page, the base address of the second page, and a count of two. If the two-byte variable occupies two successive bytes of one iB page, then, after Ac>PGS, the start and end page base addresses will be the same, and the count will be one.

The last functional is iBPAGES, used in both iBakMOVE and iBakMOVE>, whose job is to provide the correct arguments for either iBMOVE or iBMOVE>, as well as leave on the stack the data to calculate the next iteration's arguments. iBPAGES takes an address and count on the stack, and leaves the address+29 (ready for the next page, if there is one), the count-1 (this number being zero is taken as termination), and then the original address, the iB page (calculated *from* the address), and the count of 29 (iBDATA), which are used as a set of three by either iBMOVE or iBMOVE>.

With these functionals, we can construct the final words iBakMOVE and iBakMOVE>. For both words, we use Ac>PGS to calculate our looping parameters, and then begin a loop of:

```
iBPAGES iBMOVE    (or iBPAGES iBMOVE>).
```

The phrase 0 ,Y LDD, 0= UNTIL, non-destructively tests the top of stack (which holds the decrementing count), so that the loop continues until the count is zero. Finally, we drop the two remnants from the stack. So we can use these words as follows:

```
Address Count iBakMOVE
```
to write data to the iBak

```
Address Count iBakMOVE>
```
to read data from the iBak

Note that we don't need to know the associated iBak addresses or whether the array wraps over between pages—the iBakMOVE words do all of this work for us! We can move the *entire* array to and from the iBak with the following phrases:

```
SUB: RAM>iBAK   iBAKRAM iBAKSIZE iBakMOVE
;SUB
SUB: iBAK>RAM   iBAKRAM iBAKSIZE iBakMOVE>
;SUB
```

To automatically back up any variables as we write to them, we could easily develop, say:

```
SUB: C!Bak ( n A -- )  TUCK C! 1 iBakMOVE ;SUB
SUB:  !Bak ( n A -- )  TUCK  ! 2 iBakMOVE ;SUB
```

```
\ StartUp - Compilers                          DCE 13:05 03.02.98
HEX
ASM DEFINITIONS 5000 CONSTANT ROMBASE   2000 CONSTANT ROMSIZE
   ROMBASE ROMSIZE + CONSTANT ROMEND

EMULATE!
          TARGET .IF 0 ROMSIZE - THERE - .ELSE 0 .THEN dA !

: RAMBASE TARGET  IF 2000  ELSE ROMBASE 1800 -  THEN ;

 RAMBASE               RAMPTR !
 ROMBASE 40 + ( RegsArea) TDP !     THERE ROMSIZE 42 - FF FILL

\ StartUp - RamLocations, Lines              DCE 15:06 06.05.99
HEX
B02E CONSTANT SCSR          B02F CONSTANT SCDR
  7E CONSTANT JMPop

CVAR OUTJMP VAR STDOUT                      ( OutPutRedirect  )
CVAR STOPS                               ( Errors that cause STOP!)

  08 RARRAY #$        ( Formatting NumberStrings)

       VAR LOOPTR    ( LoopPointer - i.e. Counter)
       VAR LOOPND    ( LoopEnd     - i.e. Limit)


\ StartUp - Primitives                         DCE 13:04 13.05.99

SUB:   wait     BEGIN, .B CLR, BEGIN, .B DEC, 0= UNTIL,
     .A DEC, 0= UNTIL, ;SUB

SUB:  OUT       OUTJMP JMP, ;HC

   : #OUT       # LDB, OUT ;

SUB: PSHD               DEY, DEY, 0 ,Y STD, ;SUB
SUB: PSHB       .A CLR, DEY, DEY, 0 ,Y STD, ;SUB

   : CPUSH      LDB, PSHB ;
   :  PUSH      LDD, PSHD ;

SUB: POPD       0 ,Y LDD, INY, INY, ;SUB
SUB: OVER       2 ,Y LDD, PSHD ;SUB
SUB: SWAP       0 ,Y LDD, 2 ,Y LDX, 0 ,Y STX, 2 ,Y STD, ;SUB
SUB: NIP        POPD 0 ,Y STD, ;SUB

SUB: lit        PULX, (X) LDD, INX, INX, PSHX, PSHD ;SUB
   : LIT        lit , ;      ( n -- \ Compile n as Literal)

SUB: 2DROP      INY, INY, INY, INY, ;SUB        ( n n -- )
SUB: DROP       INY, INY, ;SUB                    ( n -- )

'S 2DROP DUP CONSTANT POP2 4 + CONSTANT POP

SUB: DUP        0 ,Y LDD, PSHD ;SUB    ( n -- n n )
SUB: 2DUP       2 ,Y LDD, PSHD 2 ,Y LDD, PSHD ;SUB
SUB: 3DUP       4 ,Y LDD, PSHD 4 ,Y LDD, PSHD
                4 ,Y LDD, PSHD ;SUB



SUB: ROT        4 ,Y LDD, N STD, 2 ,Y LDD, 0 ,Y LDX,
                2 ,Y STX, 4 ,Y STD, N LDD, 0 ,Y STD, ;SUB
```

FORML, from page 78.

The book includes all the code required with documentation. (R.E.Haskell,:*Design of Embedded Systems Using 68HC12/11 Microcontrollers*, Prentice-Hall, Upper Saddle River, NJ, 2000)

Of similar interest is the use of Forth (FICL comes to mind) to wrap C/C++ code, such as drivers. It could provide an interactive user interface at the top, and hardware emulation at the bottom. This would permit testing, debugging, and evaluating the driver code long before the hardware becomes available.

Philip Daunt, whose practice is law, provided a first for FORML, a talk not about Forth but on the Law. He listed some of the problems in which engineers-turned-businessmen can be ensnared.

Elizabeth Rather spoke to the request that the equivalent of MS Foundation Classes be provided in SwiftForth. The answer: they are exploring the task, but don't hold your breath. Not only is the number and complexity of these classes daunting, but not a few have bugs, as well. A useful subset may be a possibility, with the Forth community filling in the rest.

Chuck Moore provided a few more details about Color Forth. It has no name fields. Instead, words are hashed and their addresses located through a table. `Words`, then, will not provide a list of words in the dictionary.

The new single command line works well. As you type, letters enter from the right, travel left, and vanish at the left edge.. Words are interpreted as soon as a blank space is entered. You always have a history of 80 characters.

Of the short papers that wrapped up the discussion, I enjoyed most the one by Dr. C.H. Ting. He spoke to the essence of Forth, the "Tao of Forth."

Michael Ham had written, "Forth is like the Tao; it is a Way, and is realized when followed. Its fragility is its strength; its simplicity is its direction." Ting sought a greater simplification inspired by the Tao Te Ching, chapter 48:

"Do learn daily, increase. Do Tao daily, decrease. Decrease and

```
SUB: -ROT       0 ,Y LDD, N STD, 2 ,Y LDD, 4 ,Y LDX,
                2 ,Y STX, 0 ,Y STD, N LDD, 4 ,Y STD, ;SUB

SUB: +          2 ,Y LDD, 0 ,Y ADDD, 2 ,Y STD, POP JMP, ;HC

SUB: 1-         0 ,Y LDD,  1 # SUBD, 0 ,Y STD, ;SUB ( n -- n-1)
SUB: 1+         0 ,Y LDD,  1 # ADDD, 0 ,Y STD, ;SUB ( n -- n+1)

SUB: C@         0 ,Y LDX, (X) LDB, .A CLR, 0 ,Y STD, ;SUB

SUB: C!         2 ,Y LDD, 0 ,Y LDX, (X) STB, POP2 JMP, ;HC
SUB: !          2 ,Y LDD, 0 ,Y LDX, (X) STD, POP2 JMP, ;HC

HEX
SUB: M/MOD ( d n -- R Q \ Divides n into d => Rem Quotient )
                4 ,Y LDD, 2 ,Y LDX, 4 ,Y STX,
                ASLD,    2 ,Y STD, 10 # LDX,
         BEGIN, 4 ,Y LDD, .B ROL, .A ROL,
     CS NOT IF, 0 ,Y CPD, CS NOT IF, F[  SWAP ] F
                                THEN, 0 ,Y SUBD, SEC,
          ELSE, CLC, THEN, 4 ,Y STD, 3 ,Y ROL, 2 ,Y ROL,
 DEX, 0= UNTIL, POP JMP, ;HC

SUB: /MOD       0 ,Y LDD, N STD, 0 # LDD,        ( n n -- R Q )
                0 ,Y STD, N PUSH M/MOD ;SUB

SUB: T0=        POPD 0 # CPD, ;SUB          ( --cc0= if Top=0 )

SUB: COUNT      0 ,Y LDX, (X) LDB, INX, 0 ,Y STX, PSHB ;SUB

SUB: "-1"           -1 # PUSH ;SUB
SUB: "0"             0 # PUSH ;SUB
SUB: "2"             2 # PUSH ;SUB
SUB: "3"             3 # PUSH ;SUB

SUB: CMOVE ( A B c -- \ Move memory from A to B for c bytes )
         BEGIN, 0 ,Y LDX, 0= NOT WHILE, DEX, 0 ,Y STX,
                4 ,Y LDX, (X) LDB, INX, 4 ,Y STX,
                2 ,Y LDX, (X) STB, INX, 2 ,Y STX,
        REPEAT, DROP POP2 JMP, ;HC

\ FORTHKernel - Converters                    DCE 11:49 16.06.99
HEX
SUB: do         POPD LOOPTR STD,                  ( End St --)
                POPD LOOPND STD, ;SUB

SUB: loop       LOOPTR LDD, 1 # ADDD, LOOPTR STD,      ( -- CS)
                LOOPND CPD, ;SUB

   : DO,        do   BEGIN, ;
   : LOOP,      loop CS NOT UNTIL, ;

\ Variables - RAMarrays                       DCE 11:49 16.06.99
HEX
    CVAR iBCNT              ( Utility Count byte )
    CVAR iBCRC CVAR CRChi   ( CRCBytes, low, high)
    CVAR iBRETRY            ( ReTry Counter )

20 RARRAY UDPBUFF           ( UniveralDataPacket Buffer )

SUB: "UDPBUFF"    UDPBUFF # PUSH ;SUB ( -- n ) Pushes "UDP"

\ CRC16Low                                    DCE 14:54 29.07.97
HEX
```

decrease until — nothing. Do no, do and no no do."

In American idom, "To acquire knowledge, add. To gain Wisdom, subtract."

Thus, the Tao of Forth is **-- >** :

(The meaning of the Tao of Forth escaped me, so I asked Dr. Ting about it. The colon, of course, is the Colon Word. This Word represents the start of all Forth, the point at which new definitions are defined that take the essence of Forth and extend it to reach the universe.

On further refection, it is obvious that the essence—that is, the Wisdom—of Forth is derived by subtracting non-essential Words from Forth until an irreducible set of Words is left. In describing his P8 Forth processor, Dr. Ting had reduced that number of primitive Words to 25. From these, the Forth core, kernel, programming environment, and program itself eventually all flow.)

Four attendees brought hardware to demonstrate.

Dr.Ting showed the wire-wrap setup he used to program his P8 Forth processor.

John Hart demonstrated a stepper-motor controller using a PGA. By using the current pulses driving the stepper, it could determine the position of the motor without external sensors.

András Zsótér demonstrated the iTV web hardware. An old Sinclair computer provided the keyboard input, a TV the display, and a portable computer emulated a web site. Text is clear, though limited to TV resolution. The hard part is still to come—ramping up for production and devloping a market adequate to make it pay.

John Hall brought some of the colorful clamshell Apple portable computers. They resemble game machines and I see the appeal to the high school and college crowd. They are complete systems with a fast response, but are surprisingly heavy.

The meeting ended with the distribution of prizes and closing remarks by Richard Wagner.

```
HC: CRC16LO  00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             00 C, C1 C, 81 C, 40 C, 01 C, C0 C, 80 C, 41 C,
             01 C, C0 C, 80 C, 41 C, 00 C, C1 C, 81 C, 40 C, ;HC

\ CRC16Hi                                        DCE 14:54 29.07.97
HEX
HC: CRC16HI 00 C, C0 C, C1 C, 01 C, C3 C, 03 C, 02 C, C2 C,
            C6 C, 06 C, 07 C, C7 C, 05 C, C5 C, C4 C, 04 C,
            CC C, 0C C, 0D C, CD C, 0F C, CF C, CE C, 0E C,
            0A C, CA C, CB C, 0B C, C9 C, 09 C, 08 C, C8 C,
            D8 C, 18 C, 19 C, D9 C, 1B C, DB C, DA C, 1A C,
            1E C, DE C, DF C, 1F C, DD C, 1D C, 1C C, DC C,
            14 C, D4 C, D5 C, 15 C, D7 C, 17 C, 16 C, D6 C,
            D2 C, 12 C, 13 C, D3 C, 11 C, D1 C, D0 C, 10 C,
            F0 C, 30 C, 31 C, F1 C, 33 C, F3 C, F2 C, 32 C,
            36 C, F6 C, F7 C, 37 C, F5 C, 35 C, 34 C, F4 C,
            3C C, FC C, FD C, 3D C, FF C, 3F C, 3E C, FE C,
            FA C, 3A C, 3B C, FB C, 39 C, F9 C, F8 C, 38 C,
            28 C, E8 C, E9 C, 29 C, EB C, 2B C, 2A C, EA C,
            EE C, 2E C, 2F C, EF C, 2D C, ED C, EC C, 2C C,

( CRC16HI ) E4 C, 24 C, 25 C, E5 C, 27 C, E7 C, E6 C, 26 C,
            22 C, E2 C, E3 C, 23 C, E1 C, 21 C, 20 C, E0 C,
            A0 C, 60 C, 61 C, A1 C, 63 C, A3 C, A2 C, 62 C,
            66 C, A6 C, A7 C, 67 C, A5 C, 65 C, 64 C, A4 C,
            6C C, AC C, AD C, 6D C, AF C, 6F C, 6E C, AE C,
            AA C, 6A C, 6B C, AB C, 69 C, A9 C, A8 C, 68 C,
            78 C, B8 C, B9 C, 79 C, BB C, 7B C, 7A C, BA C,
            BE C, 7E C, 7F C, BF C, 7D C, BD C, BC C, 7C C,
            B4 C, 74 C, 75 C, B5 C, 77 C, B7 C, B6 C, 76 C,
            72 C, B2 C, B3 C, 73 C, B1 C, 71 C, 70 C, B0 C,
            50 C, 90 C, 91 C, 51 C, 93 C, 53 C, 52 C, 92 C,
            96 C, 56 C, 57 C, 97 C, 55 C, 95 C, 94 C, 54 C,
            9C C, 5C C, 5D C, 9D C, 5F C, 9F C, 9E C, 5E C,
            5A C, 9A C, 9B C, 5B C, 99 C, 59 C, 58 C, 98 C,
            88 C, 48 C, 49 C, 89 C, 4B C, 8B C, 8A C, 4A C,
```

```
            4E C, 8E C, 8F C, 4F C, 8D C, 4D C, 4C C, 8C C,
            44 C, 84 C, 85 C, 45 C, 87 C, 47 C, 46 C, 86 C,
            82 C, 42 C, 43 C, 83 C, 41 C, 81 C, 80 C, 40 C, ;HC


\ iButtons - Start of actual iButton code!   DCE 11:06 17.06.99
HEX

: iBDL          B000 ;          ( B000 is PortA )
: iBDm          80 ;            ( $80  is bit 7 )
: iBDR          B026 ;          ( B026 is PortA "iBDirnReg")
: iBRm          80 ;            ( $80  is bit 7 )

SUB: iBPAGE     20 LIT ;SUB             ( 32Bytes per Page )
SUB: iBDATA     1D LIT ;SUB             ( 29bytes+Cnt|CHK|SUM|)


SUB: tmwait ( INPUT in B, Uses 5n+11 Cycles)
               BEGIN, .B DEC, 0= UNTIL, ;SUB

(Macro: Computes on the ASSEMBLE-TIME stack, compiles nothing!)

: uSwait     F[  2* 15 - 5 / ]F ;  ( A:uS -- A:Wc for2MHzClock)

SUB: WRLO ( t -- ) ( Uses same time as "tmwait", but on stack!)
( SetupLowOut:) iBDL LDB, iBDm FF XOR # .B AND, iBDL STB,
( Dirn Output:) iBDR LDB, iBRm        # .B ORA, iBDR STB,
( Wait:)        0 ,Y LDD, BEGIN, .B DEC, 0= UNTIL,
( Dirn  Input:) iBDR LDB, iBRm FF XOR # .B AND, iBDR STB,
( Drop Stack  ) DROP ;SUB

             ( N.B. This is a CRITICALLY TIMED Subroutine!!! )
SUB: RDiBRST ( leaves 0= if LoSensed in 480uS after WriteLo)
               SEI,                         ( Set InterruptMask)
         ( 448) 1C0 uSwait # CPUSH WRLO     ( SendPresencePulse)
               .A CLR,                      ( Clear Result Reg)
               1D # LDB, iBCNT STB,         ( Setup Counter)
         BEGIN, iBDL LDB, iBDm # .B AND,    ( Pulled Low? )
               0= IF, 1 # LDA,              ( Yes:Setup Answer)
                ELSE, NOP, NOP,             ( ElseTime balance)
                THEN, iBCNT DEC,            ( Count Loops )
      0= UNTIL, CLI, 1 # .A CMP, ;SUB       ( Setup return flag)

SUB: iBPRESENT  RDiBRST 0= NOT IF, Error! THEN, ;SUB

SUB: !iBYTE     01 # LDB, N    STB,                 ( c -- )
         BEGIN, 0 ,Y LDD, N .B AND,
         0= IF, ( Wr0:) SEI, 3C uSwait # CPUSH WRLO  CLI,
          ELSE, ( Wr1:) SEI, ( now write lo for 3uS:)
( SetupLowOut:) iBDL LDB, iBDm FF XOR # .B AND, iBDL STB,
( Dirn Output:) iBDR LDB, iBRm        # .B ORA, iBDR STB,
( Dirn  Input:)           iBRm FF XOR # .B AND, iBDR STB,
                             34 uSwait # LDB, tmwait CLI,
          THEN, N ROL, CS UNTIL, DROP ;SUB

SUB: @iBYTE     DEY, DEY, 0 ,Y CLR, 8 # LDB, N STB, ( -- n)
         BEGIN, SEI,
( SetupLowOut:) iBDL LDB, iBDm -1 XOR # .B AND, iBDL STB,
( Dirn Output:) iBDR LDB, iBRm        # .B ORA, iBDR STB,
( Dirn  Input:)           iBRm -1 XOR # .B AND, iBDR STB,
               4 # LDB, tmwait              ( PostPulseWait)
               iBDL LDB, iBDm # .B AND,     ( LowSensed?)
         0= IF, 1 # LDA, ELSE, .A CLR, THEN, ( SetupResultReg)
               CLI, 1 # LDB, tmwait         ( PostSenseWait)
```

```
                1 # .A CMP,                    ( TestResultReg)
          0= IF, CLC, ELSE, SEC, THEN, 1 ,Y ROR, N DEC,
       0= UNTIL, ;SUB

SUB: iBSKIP     iBPRESENT CC # CPUSH !iBYTE ;SUB

SUB: RDSX ( -- TA1 TA2 E/S )
                iBSKIP AA # CPUSH !iBYTE
                @iBYTE   @iBYTE   @iBYTE ;SUB

SUB: SXOK       RDSX POPD 2DROP 20 # .B CMP,     ( ccCS=>OK)
          CS IF, CLC, ELSE, SEC, THEN, ;SUB

SUB: SXCHK ( CA TA c -- CA TA  c ALO AHI ES) ( & CS if Error)
                RDSX 5 ,Y LDD, 9 ,Y .B CMP,
     0= NOT IF, SEC,
           ELSE, 3 ,Y LDD, 8 ,Y .B CMP,
            0= NOT IF, SEC,
                ELSE, 0 ,Y LDD, 20 # .B CMP,
                      CS NOT IF, SEC, ELSE, CLC, THEN,
                THEN,
           THEN,
     CS NOT IF, A ,Y LDX, 7 ,Y CPUSH
          BEGIN, @iBYTE POPD (X) .B CMP,
                0= NOT IF, 1 # LDB, SEC,
                     ELSE, 1 ,Y LDB, .B DEC,
                           0= IF, 0 # LDB, SEC,
                            ELSE, 1 ,Y STB, INX, CLC, THEN,
                     THEN,
      CS UNTIL, DROP 1 # .B CMP, 0= IF, SEC, ELSE, CLC, THEN,
           ELSE, RDiBRST SEC,   ( do TMRST to clear iBState)
           THEN, ;SUB

SUB: SXMOVE ( CA TA c -- CA TA c )
          iBSKIP 0F #  CPUSH !iBYTE
                3 ,Y  CPUSH !iBYTE
                2 ,Y  CPUSH !iBYTE   4 ,Y LDX, DUP
          BEGIN, (X) LDB, PSHB !iBYTE 1 ,Y LDB, .B DEC,
          0= IF, SEC, ELSE, 1 ,Y STB, INX, CLC, THEN, CS UNTIL,
                DROP ;SUB


SUB: SXCOPY ( CA TA c ALO AHI ES -- )
          iBSKIP 55 #     CPUSH !iBYTE
                4 ,Y LDD, PSHB !iBYTE
                2 ,Y LDD, PSHB !iBYTE !iBYTE
                2DROP 2DROP DROP ;SUB

SUB: UDPCRC 0 ,Y LDD, iBCRC .B EOR, 0 ,Y STD,   ( n -- )
             CRC16HI # ADDD, XGDX, (X) LDB, N STB,
       POPD CRC16LO # ADDD, XGDX, (X) LDB, CRChi .B EOR,
             iBCRC STB, N LDB, CRChi STB, ;SUB

SUB: MAKE-UDP ( CA TA c -- A' TA c+3 )
             0 ,Y LDD, UDPBUFF STB, ROT "UDPBUFF" 1+ ROT CMOVE
             0 ,Y LDD, LSRD, LSRD, LSRD, LSRD, LSRD,
             CRChi STA, iBCRC STB, "UDPBUFF" DUP C@ 1+
     BEGIN, OVER C@ UDPCRC SWAP 1+ SWAP 1- 0 ,Y LDD,
  0= UNTIL, DROP CRChi LDB, FF # .B EOR,
                  iBCRC LDA, FF # .A EOR, PSHD SWAP !
             "UDPBUFF" SWAP OVER C@ "3" + ;SUB

SUB: iBMOVE? ( CA TA c -- ) ( ccCS if OK)
```

```
                SXMOVE SXCHK
     CS NOT IF, SXCOPY SXOK
          ELSE, 2DROP 2DROP 2DROP CLC, THEN, ;SUB

SUB: iBMOVE>? ( CA TA c -- ) ( cc0= if OK)
                2 ,Y LDD, LSRD, LSRD, LSRD, LSRD, LSRD,
                CRChi STA, iBCRC STB,
         iBSKIP F0 # CPUSH !iBYTE
                3 ,Y CPUSH !iBYTE
                2 ,Y CPUSH !iBYTE NIP
        "0" DO, @iBYTE DUP UDPCRC OVER C! 1+ LOOP, DROP
                CRChi LDA, iBCRC LDB, B001 # CPD, ;SUB

SUB: iBMOVE ( CA TA c -- ) ( NOTE: "c" MUST BE <=29 $1D !!!)
                MAKE-UDP         14 # LDB, iBRETRY STB,
          BEGIN, 3DUP iBMOVE?
     CS NOT IF, 2 # LDA, wait iBRETRY DEC,
                0= IF, Error! ELSE, CLC, THEN,
           THEN, CS UNTIL, DROP 2DROP ;SUB

SUB: iBMOVE> ( CA TA c -- ) ( NOTE: "c" MUST BE <=29 $1D)
           "3" + UDPBUFF LIT -ROT 14 # LDB, iBRETRY STB,
          BEGIN, 3DUP iBMOVE>? 0= IF, SEC,
           ELSE, iBRETRY DEC,
                   0= IF, Error! ELSE, CLC, THEN,
           THEN, CS UNTIL, 2DROP
                 COUNT ROT SWAP CMOVE ;SUB


\ Backup Memory (iBak) Example:

\ Variables - Start of iBAKed variables      DCE 14:34 09.06.99

DECIMAL
RAMPTR F[  @ ]F CONSTANT iBAKram   ( Start of iBAKed Variables)
           1024 CONSTANT iBAKstart ( Start of backup IN iBAK  )

( ....declare variables we wish to back up .....)

F[  RAMPTR @ 1- iBAKram - 29 / 1+ 29 * ]F   CONSTANT iBAKsize

SUB: iBAKRAM    iBAKram   LIT ;SUB
SUB: iBAKSTART  iBAKstart LIT ;SUB
SUB: iBAKSIZE   iBAKsize  LIT ;SUB

SUB:  A>PG      iBAKRAM - iBDATA / iBDATA * iBAKRAM + ;SUB
SUB: Ac>PGS     OVER + 1- A>PG SWAP A>PG TUCK - iBDATA / 1+ ;SUB

SUB: iBPAGES ( A n -- A+29 n-1 A iBA 29 )
                1- OVER iBDATA + SWAP ROT DUP
                iBAKRAM - iBDATA / iBPAGE * iBAKSTART + ( A>iB)
                iBDATA ( Count) ;SUB

SUB: iBakMOVE  ( A n -- )
  Ac>PGS BEGIN, iBPAGES iBMOVE  0 ,Y LDD, 0= UNTIL, 2DROP ;SUB

SUB: iBakMOVE> ( A n -- )
  Ac>PGS BEGIN, iBPAGES iBMOVE> 0 ,Y LDD, 0= UNTIL, 2DROP ;SUB

SUB: RAM>iBAK   iBAKRAM iBAKsize LIT iBakMOVE  ;SUB
SUB: iBAK>RAM   iBAKRAM iBAKsize LIT iBakMOVE> ;SUB
```

Object-Oriented Programming in Forth

Better Than Oberon

> Casts are one of the most error-prone facilities in C++. They are also one of the ugliest syntactically.
>
> —*Bjarne Stroustrup*

> It should be recognized that the single most important contribution towards a design's reliability is *the elimination of superfluous features and facilities, and the containment of its complexity.*
>
> —*Niklaus Wirth*

What is OOP — extensible records

There are a lot of ways to implement OOP in Forth. Each method has varying levels of complexity and capability and, most importantly, differing design goals. A programmer should not be too concerned with which method is the "best." He may choose one method for one program and another method for another program. This kind of flexibility is what makes Forth strong. In C++, for example, the programmer is pretty much tied to multiple-inheritance for all of his programs, even if single-inheritance would be suitable for some and multiple-inheritance only needed for a few. The method described in this article is based on the method used in the Oberon language (designed by Niklaus Wirth). Our method is better, however, because we have constructors and destructors, the lack of which is the most serious flaw in Oberon.

Dick Pountain, while describing ordinary records, makes the following observation [1]:

> Since field-names are just global Forth definitions in the dictionary, there is nothing to prevent the programmer using a field name from one record type to reference a record of a different type. The result would of course be garbage if fetching data and corruption of the other fields if storing data (except in the unlikely event that the field offsets just happen to be the same).

This problem of accidentally using one record's fields in another record as always been with us and was especially prevalent in assembly language. OOP seems to have been invented when somebody said, "What if we purposely constructed our records to effect that 'unlikely' event of them having the same fields in the same place? We could then write code that would work on more than one type of record without having to rewrite the code for each type!" This is the crux of OOP. A type of record is defined. Later, a new type of record is defined. This record type needs all of the fields of that first type as well as some new fields as well. Instead of just writing a new record type with all of the fields in arbitrary order, we write the new record type with the fields common to the first record type in the same order and position. The new fields are appended upon those common fields, causing the new record type to be larger. All of the code written to access the first record type will work on the second record type as well. The first record type is the base class, and the extended record type is the derived class. This is called *inheritance* and it is the primary point of OOP. Niklaus Wirth speaks on inheritance [3]:

> We recognize at this point that the ultimate innovation was data type extensibility, which unfortunately remained obscured behind the much less expressive term 'object-oriented.' Rather unfortunately, this term was accompanied by a whole new nomenclature for many already familiar concepts with the aim of perpetrating a new view or metaphor of programming at large. Thus types became classes, variables instances, procedures methods and procedure activations messages.

The second major point of OOP (encapsulation) is that code should be associated with data. Each record type has functions associated with it called *methods* or *member functions*. Some member functions never change and are coded statically (early binding). Others may get rewritten in derived classes. These are called virtual member functions. Some languages, such as Oberon, will store vectors to these functions in fields within the record. Other languages, such as C++ and Pascal, will have a single field in the record (usually at the 0th offset) which contains a pointer to an array of vectors. This array contains vectors to all of the member functions associated with that record type. It is called the VMT (Virtual Method Table). The use of virtual member functions is similar to the use of vectors in traditional Forth. The idea is that the virtual member function can be rewritten (and a new value stored in its vector variable) and the higher level code which calls the virtual member function (with a `PERFORM` of the vector) does not have to be rewritten. The programmer who changes the contents of the vector does not even have to have access to the source code for the higher level code or do a recompile of it. It will be the same code but it will act differently because it will be calling a new virtual member function during the course of its execution. This is called *late binding*.

Other than Oberon, most languages use a VMT. There are arguments both for and against the use of a VMT. A good thing about the VMT is that it saves memory inside of the record (called an *object* or an *instance*). This is because only the pointer to the VMT needs to be stored in every object, rather than vectors to all of the virtual member functions. Another good thing is that, when creating a new object, the only field which has to be initialized is the pointer to the VMT (rather than several vectors). A bad thing about the VMT is that we can not modify the vectors associated with individual objects. Any

Hugh Aguilar • haguilar@dancris.com

modification of the VMT would affect all of the objects which contain a pointer to that VMT (all of the objects of a particular class). This isn't generally a problem because the normal way to modify the vectors is to derive a new class and give it a new and slightly modified VMT. Another bad thing about the VMT is that it adds an additional level of indirection to the run-time execution of virtual member functions. With modern processors (the PowerPC) that have a lot of registers, this isn't a problem. When implementing OOP on a processor that suffers from register starvation (the Pentium), however, the VMT can be unduly slow. Most CISC processors suffer from register starvation. The problem of using a VMT on a processor with too few registers has already been discussed in *Forth Dimensions* in regard to the i21 processor [8].

We will be using the Oberon method, to store the vectors as ordinary fields in the records and to do away with the VMT. One reason is that we might want put our system on an eight-bit computer. Forth is used quite a lot for embedded controllers, where eight-bit processors with few registers are still the norm. The fact that VMTs use less memory, on the other hand, might make them the preferred method for some applications. In general, though, memory is inexpensive these days and even small systems will often have the full 64K compliment.

Our OOP is a minimalist system in the spirit of Oberon, a language which Wirth presented as being the simplest language possible which is still capable of supporting serious and large-scale work. C++ seems to go in the opposite direction, acquiring over time new features (such as multiple-inheritance) which were originally deemed to be unnecessary complications. This is despite C++ inventor, Bjarne Stroustrup, making a strong effort to keep C++ as minimalist as possible. Without his personal efforts, C++ would have probably bloated completely beyond usability a long time ago. Stroustrup [2] appreciatively quotes Jim Waldo as saying:

> Proposers of new features [to C++] should be required to donate a kidney. That would make people think hard before proposing, and even people without any sense would propose at most two extensions.

One can empathize with Stroustrup's obvious frustration with proposers of new features. It is easy to propose nifty features, but difficult to appreciate the amount of work which is involved in implementing and supporting increasingly complex designs. Furthermore, many people don't appreciate that almost all design decisions involve a trade-off between conflicting goals. Features like multiple-inheritance are an absolute requirement to some programmers but are an unwanted performance degradation to other programmers and to all programmers are yet another thing that has to be learned and understood before beginning work.

Fortunately, Forth programmers don't have to donate a kidney in order to propose a new language feature. More practically, they don't have to convince a benevolent dictator that their proposal is worthy of implementation; they just implement it themselves. If a Forth programmer's language extension has some general utility, he can e-mail the source code and a write-up to the FIG office and, most likely, get it published in *Forth Dimensions*. Best of all, the readers of *Forth Dimensions* have the choice of adding the feature to their own Forth compiler or ignoring it if they have no use for it. Grady Booch says [5], "Multiple inheritance is like a parachute; you don't need it very often, but when you do it is essential." Forth programmers have the option of wearing a parachute when they pilot an airplane but of foregoing this precaution

Figure One.

```
CLASS BNODE NOTHING  \ binary tree node

    POINTER .LEFT  \ pointer to left node
    POINTER .RITE  \ pointer to right node
    VIRTUAL .COMPARE  \ ^root_node -- -1|0|1
    VIRTUAL .INSERT   \ ^root_node -- ^found_node

    VARIABLE ^^PARENT   \ set by INSERT  (0 if root_node is found_node)
                        \ will be useful for rotating nodes in the tree

    : MAKE_ROOT  \ ^^root_node -- ^new_root     \ assumes ^root_node is 0
        SELF SWAP !     \ make SELF the new root
        SELF ;

    : LOW_INSERT  \ ^^root_node -- ^found_node   \ ^root_node is nonzero
        BEGIN  DUP @  SELF .COMPARE  ?DUP WHILE
            -1 = IF     DUP @ .LEFT
            ELSE        DUP @ .RITE   THEN      \ ^^root ^^child --
            SWAP ^^PARENT !                     \ ^^child --
            DUP @ 0= IF  MAKE_ROOT  EXIT THEN   \ there is no ^child node
            REPEAT
        @ ;                         \ return ^root_node as the found node
    PRIVATE

    : <INSERT>  \ ^^root_node -- ^found_node
        0 ^^PARENT !
        DUP @ 0= IF  MAKE_ROOT  EXIT THEN
        LOW_INSERT ;

    : <BNODE>
        0               SELF .LEFT                      !
        0               SELF .RITE                      !
       [ ']  ABSTRACT    SELF VIRTUAL_ADR .COMPARE       !
       [ ']  <INSERT>    SELF VIRTUAL_ADR .INSERT        ! ;

' <BNODE>  ' NADA  END_CLASS BNODE
```

when they ride a bicycle. C++ programmers don't.

Our OOP — a user's perspective

The OOP described in this article is called OOOP (pronounced "ope"), for "Oberon-like OOP." All code is written in Forth-83 and tested under Laboratory Microsystems' UR/Forth. Figure One is a sample class definition.

The word CLASS starts the definition of a class. The word following it is the name of the class being defined (BNODE). The word following that is the name of the base class which this class is derived from. NOTHING is the root of the entire inheritance tree and is the only class without a base class. In the beginning there was NOTHING. NOTHING has two data fields. One is called .CLASS and contains a pointer to a data structure containing information about the object's class. The other is called .KRYSHA and contains a pointer to the delegating object if there is one. We will discuss delegation later on. Because every class has NOTHING as its ultimate base class, every class has the .CLASS and .KRYSHA fields. Every time that an object is constructed, its .CLASS field is filled in with information about the class that created the object. END_CLASS ends the definition of the class and, similarly to CLASS, must be followed by the class name. Between CLASS and END_CLASS, the fields and member functions are defined as well as anything else (mostly supporting colon words) that the user wants to define in there. OOOP is not like some Forth OOP systems, such as SWOOP [6], that redefine common defining words such as VARIABLE and colon and CREATE inside of class definitions. This would be confusing to the user, to have words' meanings be context-sensitive. It also prevents the user from defining normal VARIABLE and colon words, etc., inside of a class definition. Chuck Moore [7] has said, "Let the dictionary do the decision making." OOOP follows this advice in that it does not use context-sensitivity to determine the meaning of a few words. OOOP has distinct words associated with distinct operations and it is the user's responsibility to use the right word in the right place.

CLASS leaves several parameters on the stack which are updated by the field defining words and which are consumed by END_CLASS. We have several kinds of fields available. As we list these, the parameters that we mention are the ones that the user explicitly provides and do not include the parameters provided by CLASS.

FIELD
This contains data. It takes one parameter, which is the size of the field. The field is unaligned.

VIRTUAL
This contains a vector to a late-bound member function. The field is word aligned.

STRING
This contains data. It is just like FIELD except that it adds one to the size parameter in order to provide room for the string's count byte.

INTEGER
This contains data. It takes no parameters. It creates a field of one word which is word aligned. INTEGER is like VARIABLE except that INTEGER defines word-sized fields in classes while VARIABLE defines word-sized variables statically.

DINTEGER
This contains data. It is just like INTEGER except that provides room for two words.

POINTER
This contains data. It is just like INTEGER. Using this rather than INTEGER helps with self-documentation in that it tells the reader what the data is, a pointer rather than a numeric integer.

As a convention, the user should make all field names (including virtual function names) start with a dot. In the BNODE class, .LEFT and .RITE are data fields, .COMPARE and .INSERT are late-bound member functions, <BNODE> is the constructor. We don't have any early-bound member functions or a destructor in this class (later example classes will). All member functions require the object address on the top of the stack when they are called. The object address isn't listed as one of their parameters in their stack picture comment. Within the member function, the word SELF is used to reference the object address.

The constructor must explicitly fill the virtual function vectors with pointers to colon words. Remember that the virtual functions vectors are in the object itself rather than in a VMT, so they need to be initialized every time that an object is constructed. As a convention, the user should make colon words corresponding to virtual functions be that virtual function name without the dot in front and surrounded by pointy brackets (such as <INSERT> for .INSERT). Also as a convention, the user should make the constructor colon word be the class name surrounded by pointy brackets (<BNODE> in the example). The destructor colon word should be the same except with a tilde in front of the class name (that would be <~BNODE> if we had one).

A constructor is called when an object is created. Actually, all the constructors in the inheritance chain are called, starting with the most elder class (this is always NOTHING) and working down to the class of the object being created. If a base class constructor fills some field with some data and a derived class constructor fills that same field with some data, the derived class's constructor's data overwrites that of the base class. This is especially important with virtual function vectors. Most classes' constructors initialize all of the vectors. The function pointer that actually ends up in each vector, however, is the one which the latest derivation class constructor puts there. Objects are destroyed by giving their object address to DESTRUCT. DESTRUCT calls all of the destructors in the inheritance chain, but does it in the opposite order. It first calls the destructor of the class created and works its way up to the most elder base class (NOTHING) destructor.

The constructors don't have to call MALLOC for the object, this is already done for them. Similarly, the destructors don't have to call FREE for the object, this is done after they execute. The constructors and destructors should only be concerned with initializing and uninitializing the data in the fields. Constructors are quite common since most classes have virtual functions, the vectors of which need to be initialized. Destructors are fairly rare. They are primarily used when a class has a POINTER field which contains a pointer to some object which was created by the constructor. The destructor should call DESTRUCT for that object since it is not going to

be used anymore, now that the object containing the pointer to it is being destroyed. Failure to do this will result in a memory leak.

In the code for OOOP provided with this article, MALLOC uses ALLOT to allocate its memory. FREE doesn't do anything. In actual use, MALLOC and FREE should be upgraded to allocate and deallocate memory from a heap. The implementation of a heap is beyond the scope of this magazine article and we don't provide one. MALLOC, by the way, always returns a word aligned address.

A common error for programmers is to define the colon word for a virtual function (such as <INSERT>), but to forget to fill the virtual function vector (.INSERT) in the constructor. This is because OOOP is the only OOP system that requires the programmer to manually do this. OOOP has a error-checking built-in to catch this fault. All virtual function vectors are automatically initialized with pointers to BAD_VIRTUAL. At run-time, if a virtual function which has not been initialized in the constructor is called, BAD_VIRTUAL will abort the program with an error message.

The Oberon language does not have constructors or destructors. Oberon programmers would typically write a member function called INIT (or whatever other name they might establish as a convention) in each class. Upon instantiating an object, they would then call this initialization function. They would have another function (possibly called UNINIT) which they would call just prior to freeing the memory of an object. This is not a good method because there is no way to call the constructors and destructors for the base classes of an object. Wirth got a little too minimalist when he did away with constructors and destructors and, in doing so, largely killed the chance of his language being put to use in the real world.

END_CLASS is given two parameters by the user. These are pointers to the constructor and destructor. We have a word called NADA which does nothing. This should be used if the user doesn't have a constructor or destructor for his class (we don't have a destructor for BNODE, so we use NADA). The constructor's job is to initialize the virtual vectors and any data fields that need initialization. Initializing data fields is easy since we can just use SELF and the field name to get the field address. Virtual function vectors are a little harder. If we used SELF and the field name, we wouldn't get the field address but rather we would cause the function to execute (it is BAD_VIRTUAL at this time). In order to get a VIRTUAL's field address, we give the object address (usually SELF) to VIRTUAL_ADR with the virtual function's name following it in the input stream. VIRTUAL_ADR is intended to be used only in constructors.

Sometimes we don't want to initialize a virtual address because our class is not intended to ever be used to create objects. Our class is only intended to be used as a base class for other classes which will be used to create objects. Our own class does not have any defined behavior for all of its virtual functions. BNODE is like this. .COMPARE doesn't have any defined behavior in BNODE and is intended to be defined in the derived classes. BNODE's constructor initializes .COMPARE with a pointer to ABSTRACT. If we accidentally do define an object of type BNODE and try to execute .COMPARE, for that object, ABSTRACT will abort the program an error message. A class like BNODE which has ABSTRACT virtual functions is an *abstract class*. The reason that we make BNODE abstract is that there are any number of kinds of data which a person might want to store in a binary tree. We don't want to tie BNODE to any particular kind of data. We don't have any data fields in BNODE for .COMPARE to compare. Derived classes will have data and they will have a .COMPARE that can deal with this data. We are separating the concern of how a binary tree works from the concern of storing some particular kind of data. In general, abstracting behavior is the primary point of inheritance which is the primary point of object-oriented programming.

The colon word LOW_INSERT is not an early-bound member function. It does use SELF, however. LOW_INSERT is a helper function. It must be called from member functions in order that SELF will be valid within it. As a rule, helper functions should be made PRIVATE since they can't be used outside of the class anyway. More will be said about PRIVATE later. The variable ^^PARENT, similarly, is not a field. It is a static variable just like any VARIABLE defined anywhere. It is used internally by .INSERT and LOW_INSERT. As a convention, the author uses the caret in front of a variable name to indicate that a variable contains an address. Two carets indicates that it contains an address of an address. We try not to have any more levels of indirection than this since it would get confusing.

Our class-defining word (BNODE in the example), when executed, requires a parameter which is a count of how many objects are needed. If the user only wants one object, he still has to explicitly give it a 1 parameter. If the user creates more than one object, the objects are guaranteed to be contiguous in memory so that they can be used as an array. The class-defining word returns the address of the first object in the array. If the user wants to have named objects defined at compile-time, then he can give this address to CONSTANT. If the user defines an object at run-time, then he can store this address in a data structure of some kind (possibly just a VARIABLE). The author recommends against defining objects at compile-time because this involves executing the constructor at compile-time. If the software is ever going to run on an embedded controller, then this could be a problem because the constructor is written to execute on the target machine and not on the host machine. An aspect of C++ that the author doesn't like is the use of static objects (objects not local to MAIN or any other function). The constructors for these will execute before MAIN begins executing. They presumably execute in the same order that they appear in the source code. Such objects are a problem because some debuggers (Borland's Turbo Debugger) only begin debugging with the MAIN function. If a constructor has a bug and crashes, the crash occurs before the debugger begins and the programmer doesn't know which constructor crashed or how. The reader is encouraged to avoid this mess by constructing all objects at run-time. This is true for OOOP as well as C++.

OOOP provides a couple of words called SIZE and SIZE_OF. SIZE takes an object address and returns the size of that object. SIZE_OF takes no parameters but needs to be followed in the input stream by a class-defining word name. It returns the size of any object created by that class-defining word. SIZE and SIZE_OF are primarily intended for cases in which the programmer has an array of objects and needs to increment a pointer through them. The programmer is encouraged to use SIZE rather than SIZE_OF because it is possible that the objects which he thinks are of some class are actually of a derived class. By using SIZE, the size is determined at run-time and the correct value will be used. SIZE_OF

is a purely compile-time determination. Any kind of compile-time calculation goes against the spirit of object-oriented programming. The author expects that SIZE_OF will never be used. If any reader can think of a use for it, he should tell the author.

Figure Two contains a few more classes. PERSON is a fairly simple class which contains some information describing a person. The only new concept here is that we have an early-bound member function. This is FULL_NAME. An early-bound member function, similar to a late-bound member function, takes an object address parameter. Also similarly, we don't list this object address in the stack picture comment since we are going to be accessing it with SELF rather than keeping it on the stack. A difference from virtual functions, however, is that in an early-bound function we need to explicitly bind this object address to SELF. We do this with <BIND at the very beginning. We also need to unbind it at the end of the function with BIND>. We need a BIND> prior to the semicolon and also prior to any EXIT words that we may have within the function. Don't forget BIND>! This is an easy mistake to make and

Figure Two.

```
: APPEND  \ ptr ^string -- past_ptr
    COUNT 0 DO                  \ destination_ptr source_ptr --
        2DUP C@  SWAP C!
        SWAP 1+  SWAP 1+  LOOP
    DROP ;

: STRCAT  \ ^left_string, ^right_string -- ^full_string
    OVER C@  OVER C@  + >R              \ return: count of full_string
    R@ 1+ MALLOC DUP >R                 \ return: ^full_string
    1+                              \ ^left ^right first_char_adr --
    ROT APPEND  SWAP APPEND  DROP   \ --
    R>  R>  OVER C! ;

CLASS PERSON NOTHING

    40 STRING .NAME             \ 1'st line of mailing address
    40 STRING .STREET           \ 2'nd line of mailing address
    40 STRING .CITY_STATE_ZIP   \ 3'rd line of mailing address
    INTEGER   .SSN              \ social security number
    1 FIELD   .SEX              \ 'M' or 'F'
    VIRTUAL   .TITLE

    : .FULL_NAME  <BIND  \ -- ^string  \ use FREE to deallocate it
        .TITLE  .NAME  STRCAT
        BIND> ;

    : <TITLE>  \ -- ^string
        SELF .SEX C@  ASCII M  = IF  " Mr."  ELSE  " Ms."  THEN ;

    : <PERSON>
       [ '] <TITLE>     SELF VIRTUAL_ADR .TITLE     ! ;

' <PERSON>  ' NADA  END_CLASS PERSON

CLASS EMPLOYEE BNODE            \ sorted by SSN

    POINTER .PERSONAL           \ to a PERSON object
    INTEGER .WAGE               \ hourly
    1 FIELD .LANGUAGE           \ E=English, S=Spanish, B=Both

    : COMPARE#  \ a b -- -1|0|1
        2DUP = IF
            2DROP  0                    \ 0 for a=b
        ELSE
            U< IF  -1  ELSE  1  THEN    \ -1 for a<b  or  1 for a>b
            THEN ;

    : <COMPARE>  \ ^root_node -- -1|0|1
        SELF .PERSONAL @ .SSN @    SWAP .PERSONAL @ .SSN @
```

one that will crash the computer. If you have unexplained crashes, try doing a text search on <BIND and visually checking to make sure that each one has a corresponding BIND>.

The author doesn't like early-bound functions. Originally, OOOP wasn't going to have them at all. The programmer can rarely foresee what functions may need to be redefined in derived classes. Early-bound functions prevent such redefinition, but late-bound functions leave the door open for such later redefinitions. On the other hand, early-bound functions are a little faster and use no memory within the object. This can be needed for classes that have a lot of objects defined by them. As a rule, the programmer should make all member functions late-bound and only convert them to early-binding as a part of the optimization phase at the very end of the program development.

Our next class is EMPLOYEE. We have several new concepts introduced here. For one thing, we have a base class of BNODE rather than of NOTHING. This means that we have all of BNODE's fields. We have a POINTER field called .PERSONAL which contains a pointer to an object of class PERSON. In our constructor, we create this object by calling our class-defining word PERSON. We store the resulting object address in .PERSONAL. This is done using DELEGATE rather than ! (more will be said about DELEGATE later). We also override the .TITLE virtual member function of our PERSON object with our own version. We define a word <COMPARE> whose address is stored in the .COMPARE field by the EMPLOYEE constructor. Our .COMPARE function, <COMPARE>, is accessing a field in the .PERSONAL object (the .SSN field). We have a destructor that calls DESTRUCT for the object address stored in .PERSONAL.

We have another class called EMPLOYEE_BY_ID which has EMPLOYEE as its base class. This class introduces a new field called .ID and we redefine .COMPARE to use this rather than the social security number. This kind of upgrade could happen if a company used SSNs to identify their employees and later decided to use an internal employee identification number. The order that the constructors are called is important in regard to the .COMPARE field. First is BNODE's constructor which sets .COMPARE to ABSTRACT. Secondly is EMPLOYEE's constructor which sets .COMPARE to code dealing with the .SSN field. Thirdly is EMPLOYEE_BY_ID's constructor which sets .COMPARE to code dealing with the .ID field. Since this is the last constructor called, this is the code that .COMPARE refers to in EMPLOYEE_BY_ID objects.

Multiple Inheritance — we use true delegation instead

Neither Oberon or OOOP has multiple-inheritance. Given the current design of OOOP, it would be pretty much impossible to implement. All of the member functions for any base class would expect their fields to start at index zero. Multiple base classes can't all start at index zero. Multiple-inheritance would be nice, though. In our example, our EMPLOYEE class currently has BNODE as its base class and has a pointer (.PERSONAL) to an object of class PERSON as one of its fields. This pointer gets initialized by the constructor, which calls the constructor for PERSON to create a PERSON object and then stores the object address of this object in .PERSONAL. With multiple-inheritance, EMPLOYEE would have two base classes: BNODE and PERSON. This would simplify things since we would

```
        COMPARE# ;

    : <TITLE>  \ -- ^string
        KRYSHA .LANGUAGE C@  ASCII S = IF
            SELF .SEX C@  ASCII M = IF  " Sr."  ELSE  " Sra."  THEN
        ELSE
            <TITLE>  THEN ;     \ this is the <TITLE> defined in PERSON

    : <EMPLOYEE>
        1 PERSON DUP    SELF .PERSONAL              DELEGATE
        [ '] <TITLE>    OVER VIRTUAL_ADR .TITLE     !
                        DROP
        [ '] <COMPARE>  SELF VIRTUAL_ADR .COMPARE   ! ;

    : <~EMPLOYEE>
        SELF .PERSONAL @    DESTRUCT ;

' <EMPLOYEE>  ' <~EMPLOYEE>  END_CLASS EMPLOYEE

CLASS EMPLOYEE_BY_ID EMPLOYEE   \ sorted by ID

    INTEGER .ID             \ internal identification number

    : <COMPARE>  \ ^root_node -- -1|0|1
        SELF .ID @    SWAP .ID @
        COMPARE# ;

    : <EMPLOYEE_BY_ID>
        [ '] <COMPARE>  SELF VIRTUAL_ADR .COMPARE   ! ;

' <EMPLOYEE_BY_ID>  ' NADA  END_CLASS EMPLOYEE_BY_ID
```

not have to manually construct a PERSON object in our EMPLOYEE constructor (base classes' constructors are automatically called by a constructor). Also, we would not have to manually do double indirection to get at the PERSON fields (as seen in the EMPLOYEE version of <COMPARE>). This tends to clutter the code. It is also difficult for the user to remember when it is needed, which is a common source of bugs in object-oriented programming.

At one time, prior to implementing multiple-inheritance (in version 2.0 of C++), Stroustrup experimented with something called *delegation*. This is essentially what we are doing in OOOP with having a pointer to an object (.PERSONAL). The user would list the delegation classes next to the base class name in the class declaration. The construction of the delegated object would be performed automatically. From a practical standpoint, however, it doesn't really matter if he manually constructs this delegated object in his constructor or if the compiler automatically generates this code for him. Stroustrup says this about his experiment [2]:

> Unfortunately, every user of this delegation mechanism suffered serious bugs and confusion. Because of this, the delegation was removed from the design and from the Cfront that was shipped as Release 2.0. Two problems appeared to be the cause of bugs and confusion:
>
> #1 Functions in the delegating class [EMPLOYEE] do not override functions of the class delegated to [PERSON] .
>
> #2 The function delegated to [a member function of PERSON] cannot use functions from the delegating class [EMPLOYEE] or in other ways "get back" to the delegating object.
>
> Naturally, the two problems are related. ... In retrospect, I think the problems are fundamental. Solving the problem #1 would require the virtual function table [VMT] of the object delegated to be changed when it is bound to a delegating object. This seems out of line with the rest of the language and very difficult to define sensibly. We also found examples where we wanted to have two objects delegate to the same "shared" object. Similarly, we found examples where we needed to delegate through a [pointer to a base class object] to an object of a derived class [of that base class].

Stroustrup is hamstrung by his use of a VMT. He can't change the virtual function vectors for an object delegated to because they are in the VMT and every other object of that class would be affected. We *can* change these vectors. In the constructor for EMPLOYEE we use VIRTUAL_ADR to plug a new value into the vector for .TITLE which is a field in the PERSON class object pointed to by .PERSONAL. By doing this, we solve Stroustrup's #1 problem in that a function in EMPLOYEE is overriding a function in PERSON. Our new function "gets back" to the delegating object (of class EMPLOYEE) when it accesses the .LANGUAGE field there. This solves Stroustrup's #2 problem. The functions in the object being delegated (.TITLE in the object pointed to by .PERSONAL) needs to know the object address of the object that is doing the delegating (the EMPLOYEE object). This is what the .KRYSHA field is for. Every object has a .KRYSHA field. The PERSON object pointed to by the .PERSONAL pointer in the EMPLOYEE object has the object address of that EMPLOYEE object in its .KRYSHA field. When we filled the .PERSONAL pointer in our EMPLOYEE object with the object address of a PERSON object, we used DELEGATE. DELEGATE, in addition to filling this pointer (which ! could have done), also sets the .KRYSHA field in the PERSON object to point to the EMPLOYEE object doing the delegating. In our EMPLOYEE constructor, we also overrode one of the virtual vectors in the PERSON object (the .TITLE field). We filled this vector with a function that we had just written. This is a virtual function of PERSON, so its SELF is the PERSON object. Our function uses SELF to access the .SEX field in the PERSON object. Within this virtual function, it can use KRYSHA to obtain the object address of the delegating object (the EMPLOYEE object). Our function does this to "get back" to the .LANGUAGE data field in the EMPLOYEE object.

The word *krysha* is Russian and literally means "roof." In general usage, a person's krysha is an upper echelon figure in the police or the mafia who will protect that person from harm [9]. The EMPLOYEE object is the krysha of the PERSON object because the only way to get access to the PERSON object is by going through the EMPLOYEE object. The EMPLOYEE object could customize the virtual functions of the PERSON object because every use of the PERSON object would be in the context of it being a delegated object of the EMPLOYEE object. Modifying these virtual vectors does not affect other objects of PERSON class (who may or may not have a krysha) that are also in use at this time. They have their own .TITLE vectors inside of themselves set by the PERSON constructor. If we were using a VMT, then all PERSON objects would use the same .TITLE vector in the VMT, and modifying it would affect all of them. Because we don't have a VMT, we can use delegation effectively and we do not have to let the multiple-inheritance genie out of the bottle. Multiple-inheritance is nicer looking syntactically than delegation, but it is complicated to implement and slow to execute. It is clearly not in line with our minimalist Oberon-like philosophy. On the other hand, we can't ignore the concept entirely. We need to have a workable alternative — and we do.

Run-time type checking — needed for copying objects

One of Oberon's most powerful is run-time type checking. We have this too. The word IS_A takes an object address as a parameter and has a class name in the input stream after it. IS_A returns a flag indicating if that object is a member of that class. The flag will also be returned true if the object is a member of a class derived from the indicated class. The programmer should use IS_A sparingly. Bjarne Stroustrup has this to say [2]:

> RTTI [run-time type information] can be used to write thinly disguised switch statements [see Figure Three]. I have heard this style described as providing "the syntactic elegance of C combined with the run-time efficiency of Smalltalk," but that is really too kind. The real problem is that this code does not handle classes derived from the ones mentioned correctly and needs to be modified whenever a new class is added to the program. Such code is usually best avoided through the use of virtual functions. … For many people trained in languages such as C, Pascal, Modula-2, Ada, etc., there is an almost irresistible urge to organize software as a set of switch statements. This urge should most often be resisted.

Stroustrup was actually referring to the TYPEID function in C++ which provides a code for each class but which, unlike our IS_A function, does not indicate if an object's class is derived from some other class. The C++ TYPEID function corresponds to our .CLASS field All objects have this field and it contains a unique identifying number for the class of that object. Our IS_A is a lot more useful than C++'s TYPEID and can often obviate the code modifications which Stroustrup is warning against. Nevertheless, Stroustrup is right. There are some valid uses of SWITCH statements (sometimes called CASE statements), but they are the most dangerous language feature to be found within the Structured Programming paradigm. Using them is like letting the camel stick his nose inside of your tent; pretty soon you have the whole camel. Your program is no longer structured even though you may insist that you have only used Structured Programming language features. And it is not just object-oriented programming that can get fouled up, either. Those giant SWITCH statements used to implement state machines in pseudo-multitasking are a horrible thing as well. Chuck Moore has spoken out against the use of SWITCH statements. His proverb [7] is, "Let the dictionary do the decision making." He is saying that people should not pass a parameter into a function and then have that function test the parameter at *run-time* and branch to various code based upon the value of the parameter. It is better to have separate functions compiled to contain those various pieces of code. Each function is referenced by its name at *compile-time*. Both language designers seem to be seeing eye-to-eye on the subject of SWITCH statements. An observer would never guess this by examining typical Forth and C++ programs and counting the number of times that SWITCH statements are used in them (a lot for C++). For all of the adulation that is heaped upon Stroustrup by the C++ community, nobody seems to be paying attention to what he is actually saying.

Run-time type checking is most valuable when copying objects. The word OCOPY takes two parameters, a source object address and a destination object address. The source object is copied on top of the destination object if the source object's class is of the destination object's class. OCOPY uses IS_A internally. If the source object's class is a derived class of the destination object's class, then the data is truncated when it is copied. It is illegal to go the other way, from an object of a base class to an object of a derived class. The derived class has more fields than the base class and there would be no way to know what data to put in these fields. Niklaus Wirth has this to say [3]:

> The essence of a language featuring strong typing is that the type of the expression on the right-hand side of ":=" must be *assignment- compatible* with the type of the designator on the left-hand side. … [When assigning an object of a base class to an object of a derived class] there is not enough information to unambiguously specify [the result]. Such an assignment is *illegal in Oberon*. An attempt at an artificial definition, such as '[the extra fields] remain unchanged' cannot be reconciled with the axiom of assignment.

Beyond allowing assignments between objects of exactly the same class, it is best to only allow assignments from a derived class to a base class and to truncate the extra fields in the data during the copy. This is known as restrained type casting. This is all that OCOPY allows. If OCOPY is used to copy an object to an object of a derived class, it will abort with an error message. It will also abort if the source and destination objects aren't in the same inheritance chain at all (neither is a base class of the other). The programmer can give his source and destination object address to COULD_OCOPY rather than OCOPY. COULD_OCOPY will return a flag indicating if it is a legal operation that OCOPY will accept. Use of COULD_OCOPY will allow the programmer to make a more graceful exit than to just abort the program with an error message as OCOPY would do. Niklaus Wirth says this [3]:

> *Only those fields that comprise [the destination class] participate in the assignment.* Therefore it is assured that there always exists a one-to-one correspondence [between the left and right sides of the :=]. This definition has an analogy in mathematics: the projection of a higher-dimensional vector onto a lower-dimensional space. Using this analogy, we say that *the assignment is a projection of [the source type] onto [the destination type].*

Bertrand Meyer, the inventor of the Eiffel language, also speaks out against unrestrained type casting [4]:

> Typing, if taken seriously, also means that there is no way to bypass the type system. Many languages which claim to be statically (sometimes even "strongly") typed also allow developers to cheat the type system, enticing them into sordid back-alley deals sometimes known as *casts*.

Strong words these are. Clearly he is referring to C++; the hallmark of C++ programming is unrestrained type casting. Surprisingly, however, the inventor of C++, Bjarne Stroustrup, largely agrees [2]:

> The DYNAMIC_CAST operator [essentially the same as our OCOPY] serves the majority of needs I have encountered. I consider DYNAMIC_CAST to be the most important part of the RTTI [run-time type information] mechanism and the construct users should focus on.

Is unrestrained type casting really so bad? In some cases, no. A good example is numerics. When copying data from a

Figure Three.

```
void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
        // do nothing
    }
    else if (typeid(r) == typeid(Triangle)) {
        // rotate triangle
    }
    else if (typeid(r) == typeid(Square)) {
        // rotate square
    }
    // ...
}
```

simple class to a complicated class, we have easy and obvious rules for filling out the data of a complicated class. Examples would include casting an integer to a real or a real to a complex. Notice, however, that our simple class is not a base class to our more complicated class. For example, a complex number is not just a real with an extra field (the imaginary component) tacked on. A complex number has all of its arithmetic operations redefined. Assuming that these are virtual functions in the real number class, all of them would have to be rewritten. This kind of wholesale redefinition is not in the spirit of inheritance since nothing is being inherited. This is a good example of polymorphism, since both classes have member functions with the same names. It is not an example of inheritance, however.

Unrestrained type casting can be seen to be useful for a fairly narrow range of problems. It is useful when one class is a superset of the other class (complex numbers contain all real numbers) but does not inherit anything from this subset. Unrestrained type casting should not be used when the superset inherits functionality from the subset. `OCOPY` will abort in this case. Users of OOOP are encouraged to work within the constraints of `OCOPY` and to not bypass it. Let restrained type casting be the hallmark of OOOP programming! To a large extent, OOOP is being put into the public domain as an experiment to see if restrained type casting will work in the real world. Try writing large programs using OOOP. Is it ever necessary to cast from base classes to derived classes? When is the `IS_A` function ever needed? The author of OOOP would like some feedback on these questions.

Uniform Access — better than Eiffel

C++ programmers know that they should not declare data fields as being public because this makes it difficult to upgrade the processing of the data in derived classes. Given a data field X, they will generally make it private and then provide public functions called `GET_X` and `PUT_X` which fetch a datum and store a datum respectively. This really shows up what a weak language C++ is. Normally the assignment statement (=) is used for storing data into a variable. Now, however, we have a function (`PUT_X`) doing it by wrapping its parenthesis around the expression that otherwise would have stood alone on the right hand side of the assignment. Our expressions are also filled with `GET_X` function calls, each with a set of empty parenthesis. Our C++ assignments now have more parenthesis then LISP statements, and none of these parenthesis provide *any* information at all! We also have the characters "`GET_`" prefixed to all of our field names in the expression. This isn't providing the reader with any information either. These `GET_X` and `PUT_X` functions are syntactical abominations — one can hardly imagine a more thorough way to clutter up one's source code. Not only is the source code cluttered, but the object code suffers as well. Function calls of member functions are big and slow compared to direct access of data fields. Even the best optimizing compiler is going to choke on all of these `GET_X` and `PUT_X` function calls.

Let us turn away from this madness and read what Bertrand Meyer, the inventor of Eiffel, has to say [4]:

> An important property applies to feature calls written in dot notation and used as expressions: the notation is exactly the same for a Call involving a [member] function with no arguments and one involving an attribute [data field]. So the expression
>
> `P1.AGE`
>
> where entity P1 is of type `PERSON`, is applicable both if the feature `AGE` of class `PERSON` is an attribute or if it is a function.
>
> If `AGE` is an attribute, every instance of `PERSON` has a field which gives the value of `AGE` for the instance. If `AGE` is a function, that value is obtained, when requested, through some computation, presumably of the difference between the current date and a "birth date" field.
>
> For a client containing the above call, however, this makes no difference.
>
> This property of *uniform access* facilitates smooth evolution of software projects by protecting classes from internal implementation changes in their suppliers.

Uniform Access allows us to get rid of all of our `GET_X` functions. We can make our data fields public and use them in expressions. If we ever need to "smarten up" these features (such as with the calculated age in Meyer's example), we can rewrite them as functions. All of our code which uses these features can remain unchanged because the syntax for calling a member function or for accessing a data field is the same.

In C++, we couldn't upgrade a data field to a member function because data fields use a different syntax than member functions (functions require parenthesis) and so every reference to that feature would need to be located in a text search and changed. This text search could span dozens of files. If polymorphism is in use, a simple text pattern-matching search is going to find a lot references that don't need to be changed. The user has to visually inspect each text search hit to determine if it is code that needs to be changed. Massive text search and replace done on source code is an invitation to disaster. The avoidance of this is what prompted the C++ programmers to use the `GET_X` technique which, frankly, is just an invitation to a different disaster. C++ is a flexible language in regards to the problem of upgrading data fields to member functions. You *will* hang yourself, however, you have a choice of which rope to use.

Bertrand Meyer is clearly on the right track with his Uniform Access. However, he hasn't done anything about the need for `PUT_X` functions. Eiffel doesn't allow function calls on the left hand side of the assignment. Nor does C++ or any other such language. Can OOOP do this? Yes, it can! Unfortunately, the author of OOOP can't take credit for this — Forth has had true Uniform Access since the day that it was invented. Leo Brodie [7] has discussed the concept, although he didn't call it Uniform Access (his example of counting the red and green apples).

Any Forth variable can be thought of as a word that provides an address where some data can be fetched from or stored to. It is very possible to write a colon word which similarly provides an address. Both variables and colon words are called with the same syntax (just a reference to their name), so it is possible to rewrite a variable as a colon word. All code that accesses the variable, whether for storing to or fetching from, will now access the colon word without having to be modified.

Forth has always had true Uniform Access, but a lot of Forth programmers are unaware of it. Only now, with the advent of object-oriented programming, has it become important. The programmer can define a data field in his class

using FIELD or one of the words derived from FIELD (INTEGER, POINTER, etc.). Later on, he can rewrite this word as a colon word (early-binding) or as a VIRTUAL field (late-binding). All code, whether member functions or external functions, that accessed the field will now access the function. This works for both storing to and fetching from the address. Anybody who uses OOOP should be aware of Uniform Access and make use of it.

Polymorphism & information-hiding — weakly supported

There seems to be some confusion of definitions of polymorphism. Rick Van Norman says this [6]:

> *Polymorphism* goes a step further than inheritance. In it, a new subclass [derived class] inherits all the members of its parents [base classes], but may also redefine any DEFER: [virtual] members of its parents.

This isn't polymorphism, this is just inheritance as OOOP provides and as we have been describing throughout this article. Two classes can have fields of the same name that do different things *only if they are in the same inheritance chain*. That is, one is a base class of the other. In true polymorphism, two classes can have fields of the same name that do different things even though neither is a base class of the other (and the fields may not have the same relative position within the objects).

OOOP does not support true polymorphism. We have object addresses being passed around on the parameter stack and being stored in data structures. We can tell at run-time what class an object is (by using IS_A or COULD_OCOPY). There is no inherent way, however, to tell at compile-time what class an object address is of. This information would be necessary for the compiler to modify the dictionary search such that, when a field name is later referenced, the compiler uses the field name associated with the class which this object is of. The only way for the compiler to obtain this information is for the programmer to explicitly tell the compiler. In SWOOP [6], this is accomplished by tagging each use of an object address with the word USING followed by the class-defining word name. The author of OOOP has no intention of doing anything like this in OOOP. The source code would get so cluttered with all of those USING xxx tags that it would be unreadable. Most of the time, they are not needed anyway, because it is fairly rare to have classes with common field names. Generally, this happens by accident because there are a lot of classes and name-space pollution has become a problem. Name-space pollution is not unique to object-oriented programming; any large program will suffer from it. The solution, within OOP and without, is to be alert to redefinition warnings and to think up new names as needed. It is fairly rare for the programmer to purposely use polymorphism. The only example that the author can think of is numerics (the same example that we used for unrestrained type casting). Two classes, such as COMPLEX and REAL, may not be in the same inheritance chain but may have common field names (the arithmetic operations). Since OOOP is not supporting unrestrained type casting, there is little need to support true polymorphism either.

OOOP will probably never have true polymorphism. A little information hiding could be useful, though. Name-space pollution becomes a problem as programs become large. OOOP has a facility for dealing with this. Word name fields have several flag bits in them. Everybody is familiar with the 'immediate' flag bit and knows that the word IMMEDIATE sets this in the last word defined. Some Forth systems also have a few name field flag bits undefined which the user can define for his own purposes. We have a word called PRIVATE which is like IMMEDIATE except that instead of setting the 'immediate' flag bit, it sets a different flag bit which we shall call the 'private' flag bit. We have another word called END_MODULE which traverses the entire dictionary and removes any words that have their 'private' flag bit set from the dictionary search. These words are still in the dictionary and any words which call them will still work. It is just that the words can't be found in a future dictionary search and hence can't be called from any future words. If a new word is defined with the same name, there will be no redefinition warning given. In a large program, there can be thousands of words defined. Only a fraction of these are documented and intended to be used throughout the program, the rest were just of local interest and are just clutter. END_MODULE could be incorporated into END_CLASS in order to emulate the concept of private member functions as provided by C++. The author prefers to not do this. It is best to organize the code into modules (each stored in a separate file) and to use END_MODULE at the end of each one to hide all of the words in that module which are not intended to be used outside of the module. If an inheritance chain of classes are all defined within a single module, placing END_MODULE at the end of that module has the effect of making the private words act similarly to the PROTECTED words of C++.

PRIVATE and END_MODULE are very handy words for reducing name-space clutter. Because they are not strictly tied to object-oriented programming, the user has some flexibility with them. If END_MODULE is incorporated into END_CLASS, then PRIVATE words are essentially the same as C++ PRIVATE words. If END_MODULE is used at the end of each module, then PRIVATE words are essentially the same as STATIC words in C and C++ modules. If an entire inheritance chain is in a single module, then PRIVATE words are essentially the same as C++ PROTECTED words. Despite being a fairly simple construct, our private concept works fairly well. It is not as thorough as C++'s PUBLIC, PROTECTED, and PRIVATE words (especially if one takes into account how C++ classes also declare their base classes as PUBLIC, PROTECTED, or PRIVATE). On the plus side, it works quite well and it doesn't involve a lot of source code clutter like the USING xxx tags of SWOOP. The only real problem with PRIVATE and END_MODULE, is that the author doesn't know of any way to write them in Forth-83. He does know how to write them in UR/Forth, which is the compiler that he uses. The implementation, however, involves accessing internal non-standard features of UR/Forth. The OOOP code provided with this magazine article has only dummy definitions of PRIVATE and END_MODULE. The reader needs to write these himself so that they will work with whatever compiler he is using. If the reader doesn't know how to do this, he is still encouraged to use PRIVATE and END_MODULE throughout his application in the hopes that he will eventually be able to put some substance to these words.

Implementation — nothing very complicated

The complete code for OOOP is in the file OOOP.4TH which is provided with this article. It is straightforward and uncomplicated Forth-83 code. The reader should be able to quickly get it running on any Forth-83 system and probably,

with a little more effort, on any ANSI Forth system. OOOP is currently running under UR/Forth. If anybody does port it to any other Forth system, please e-mail a copy to the author.

There are some sticky points. The PRIVATE/END_MODULE pair, which we have already discussed, is the worst. Another area in which the implementor is going to need to do some custom programming is MALLOC and FREE. MALLOC uses ALLOT internally. This is not suitable for practical use. The reader will need to rewrite MALLOC and FREE to use a heap. Heaps aren't complicated, but they are beyond the scope of this article. There are some design decisions that the implementor needs to make. Mostly, he needs to decide if he is going to have a static internal array which he allocates memory from (good on an embedded controller), or if he is going to use the operating system's memory allocation facility (good on a desktop computer). Also complicating the implementation would be if the programmer wants to get involved in using far pointers on a processor that has banked memory (such as the 8086). If there is interest in how to implement a heap, readers should contact the author. Perhaps we can have a *Forth Dimensions* article about heaps sometime in the future.

A minor point that may cause implementors difficulty is the COMPILE word. A very grievous weakness of Forth-83 is that we don't have any standard word which takes a CFA and compiles it. This word is easy to write (on a threaded system it is just a comma), however, it is different on every compiler. This kind of situation is exactly what standardized words are for — they hide compiler specific details from the application programmer. The result of this weakness is that we don't have a good way to write macros (immediate words that compile a sequence of words). Within the sequence of words, we have to deal with immediate words differently than with non-immediate words. We use [COMPILE] for immediate words and COMPILE for non-immediate words. We can use WORD FIND to determine if a word is immediate or non-immediate. Once we have used WORD to take it out of the input stream, however, we have no way of compiling it since COMPILE needs it *in* the input stream. We don't have a word which can take the CFA provided by FIND and compile it. Essentially, COMPILE needed to be factored into its two constituent parts: getting the word out of the input stream and compiling it.

OOOP has a few macros (SELF, <BIND, and BIND>). These use COMPILE and assume that the words being compiled are all non-immediate. We have a word called CHECK_IMMEDIACY that is run at compile-time to check that the words used by COMPILE actually are non-immediate. If they are immediate on any compiler that the reader may be using, then the reader will have to rewrite the macros. >R and R> are words that, in some Forth compilers, are immediate and, in other Forth compilers, are non-immediate. >R and R> may cause problems. The difficulty in writing macros is one of those heartbreak situations where Forth-83 *almost* does what is needed.

Other than these points, OOOP should be easy to implement on any compiler. We don't use any particularly fancy programming techniques, such as dictionary search modifications (vocabularies) or "second order defining words" [1]. OOOP should be easily implemented on a cross-compiler. This is important, since we have said that OOOP is intended for use on embedded controllers, and these are generally programmed with cross-compilers. The author hasn't tried implementing OOOP on a cross-compiler. The author has written a cross-compiler (MFX for the MiniForth processor at Testra), however, and is quite familiar with how cross-compilers work. The author predicts no difficulty in getting OOOP to run on MFX. Any cross-compiler comparable to MFX in capability should also work just fine.

Summary — keep it simple

Forth's primary arena is embedded controllers, many of which are eight-bit. OOOP is designed to be used in this arena. As such, it has been purposely kept as simple as possible. This has several benefits. One is easy implementation — OOOP is going to have to be implemented by programmers who are eager to get started on their application and don't really want to delve into any systems-level Forth programming. A lot of these programmers may actually be electrical engineers who see programming as being a small part of the work involved in a project and would like it to be smaller. Another benefit is reliability. As Wirth pointed out in the quote at the top of the article, simplicity is the key to reliability. Because OOOP is "pure" object-oriented programming assuming that the programmer uses restrained type casting), OOOP programs are amenable to verification of their correctness. This can be important in embedded systems that control machines which, if they fail, could put people in the hospital or smash up expensive equipment. Yet another benefit of OOOP is a reduced learning curve. If a feature isn't going to be used by most of the users, then all of the users should not be forced to learn it. OOOP does not have any superfluous or gratuitous features, and the features that it does have are uncomplicated.

OOOP is not necessarily the best choice for all applications. As projects get bigger, polymorphism and information hiding become increasingly important. OOOP doesn't have true polymorphism. Our PRIVATE mechanism provides somewhat crude information hiding in that it only simulates the creation of protected words if all the class definitions of an inheritance chain are in a single module. Desktop software tends to get a lot bigger than embedded controller software. Desktop software also usually involves supporting a large API (set of interface words to the operating system). Name-space conflicts can become common. OOOP, which doesn't do any modification of the dictionary search, is probably not a good solution in this environment. In general, programmers should choose their OOP system depending upon the application which they are working on. The strength of Forth is that choices like these can be made. If a programmer doesn't make these choices but expects to be given a single standard solution to use on every application, then he is not taking advantage of the Forth language. It is a mistake to become trapped in the idea that there is a single *best* solution waiting to be found. This is the path to mediocrity, because no single solution is going to be optimum for the entire spectrum of applications that can be written. Ironically, the path to mediocrity is most heavily traveled by elitist types who feel that they deserve only the *best*.

Bibliography

[1] *Object Oriented Forth*, Dick Pountain, 1987
[2] *The Design and Evolution of C++*, Bjarne Stroustrup, 1994
[3] *Programming in Oberon — Steps beyond Pascal and Modula*, Martin Reiser and Niklaus Wirth, 1992
[4] *Eiffel: the language*, Bertrand Meyer, 1992
[5] *Object-Oriented Design*, Grady Booch, 1991
[6] "SWOOP: Object-Oriented Programming in SwiftForth", *Forth Dimensions* (XX.5,6), Rick Van Norman.
[7] *Thinking Forth*, Leo Brodie
[8] "DynOOF-style Objects for the i21 microprocessor", *Forth Dimensions* (volume XX, number 4), András Zsótér
[9] *Dermo! The Real Russian Tolstoy Never Used*, Edward Topol, 1997

Listing One.

```
\ OOOP.4TH -- Oberon-like Object Oriented Programming system
\ by Hugh Aguilar


\ ****** Preliminary code ******

WSIZE 4 = .IF       \ 32-bit system

    4 CONSTANT W

    : W+    4 [ COMPILE] LITERAL  COMPILE + ;    IMMEDIATE
    : W-    4 [ COMPILE] LITERAL  COMPILE - ;    IMMEDIATE
    : W*    COMPILE 2*  COMPILE 2* ;             IMMEDIATE
    : W/    COMPILE 2/  COMPILE 2/ ;             IMMEDIATE

    \ all of these should be in assembly language

.THEN

WSIZE 2 = .IF       \ 16-bit system

    2 CONSTANT W

    : W+    COMPILE 2+ ;    IMMEDIATE
    : W-    COMPILE 2- ;    IMMEDIATE
    : W*    COMPILE 2* ;    IMMEDIATE
    : W/    COMPILE 2/ ;    IMMEDIATE

.THEN

-1 CONSTANT TRUE
 0 CONSTANT FALSE

: BAD_VIRTUAL  \ --
    TRUE ABORT" *** The constructor didn't fill in this VIRTUAL's vector. *** " ;

: NEEDED  \ adr -- offset       \ offset to add to adr to make an aligned address
    W MOD >R  R@ IF
        W R> -
    ELSE
        R>  THEN ;

: ALIGNED_HERE  \ -- here_value     \ calls ALLOT with value [ 0,W)
    HERE NEEDED ALLOT
    HERE ;

: MALLOC  \ size -- adr
    ALIGNED_HERE >R                 \ size --   \ aligned adr on return stack
    DUP ALLOT                       \ size --   \ allocate memory at R@ adr
    R@ +  R@  ?DO                   \ --
        [ '] BAD_VIRTUAL  I !
        W +LOOP
    R> ;

\ MALLOC fills the data with vectors to BAD_VIRTUAL.  Because VIRTUAL always
\ aligns the vectors, the vectors will all be initialized with BAD_VIRTUAL.
\ If a user forgets to properly initialize a vector in the constructor, the
\ first time that this member function is called, BAD_VIRTUAL will execute.
```

```
: FREE  \ adr --
    DROP ;

\ For simplicity we use ALLOT in MALLOC.  This should be changed to use a heap
\ if it is desired to be able to deallocate nodes.  FREE would then be changed
\ to deallocate the memory in the heap.

: PFA  \ -- pfa  \word: structure_name
    ' >BODY ;

: PFA_FIELD  \ index --
    DUP 0= IF  DROP
        : [ COMPILE] ;  IMMEDIATE   \ don't waste run-time adding zero to the pfa
        EXIT THEN
    CREATE
        W* ,        \ store the offset for use by DOES>
    DOES>  \ pfa -- field_adr
        @ + ;

\ PFA and PFA_FIELD are a primitive way to access fields within a structure
\ (any word defined with CREATE).  OOOP is a much more sophisticated method,
\ but we need PFA and PFA_FIELD for writing OOOP.

: PRIVATE  \ --         \ make the last definition private
    ;

: END_MODULE  \ --      \ remove all private definitions from dictionary search
    ;

\ PRIVATE and END_MODULE can't be written in Forth-83 that I am aware of (I know
\ how to write them in UR/Forth).  PRIVATE should set a bit in the name field
\ (similar to how IMMEDIATE works).  END_MODULE should traverse the entire
\ dictionary and remove all of the words that have their private bit set from
\ the dictionary search.  These words will still exist in code memory and will
\ execute at run-time when the words which call them are executed.  They can't
\ be found in the dictionary, however, and so can't be executed from the keyboard
\ or compiled into any future words.


\ ****** Binding ******

VARIABLE <SELF>     \ this is the current object

: CHECK_IMMEDIACY  \ --
    " +"        FIND NIP  -1 <> ABORT" *** +        should be non-immediate ***"
    " -"        FIND NIP  -1 <> ABORT" *** -        should be non-immediate ***"
    " 2*"       FIND NIP  -1 <> ABORT" *** 2*       should be non-immediate ***"
    " 2/"       FIND NIP  -1 <> ABORT" *** 2/       should be non-immediate ***"
    " 2+"       FIND NIP  -1 <> ABORT" *** 2+       should be non-immediate ***"
    " 2-"       FIND NIP  -1 <> ABORT" *** 2-       should be non-immediate ***"
    " >R"       FIND NIP  -1 <> ABORT" *** >R       should be non-immediate ***"
    " R>"       FIND NIP  -1 <> ABORT" *** R>       should be non-immediate ***"
    " @"        FIND NIP  -1 <> ABORT" *** @        should be non-immediate ***"
    " !"        FIND NIP  -1 <> ABORT" *** !        should be non-immediate ***"
    " <SELF>"   FIND NIP  -1 <> ABORT" *** <SELF>   should be non-immediate ***"
    ;
CHECK_IMMEDIACY  \ verify the COMPILE words used in various places

: SELF  \ --                                \ compile-time
    \ -- object_adr                         \ run-time
    COMPILE <SELF>  COMPILE @ ;  IMMEDIATE
```

```
: <BIND  \ --                                  \ compile-time
    \ object_adr --                            \ run-time
    COMPILE <SELF>  COMPILE @  COMPILE >R       \ push old <SELF> to return stack
    COMPILE <SELF>  COMPILE ! ;  IMMEDIATE      \ set <SELF> to object_adr

: BIND>  \ --                                  \ compile-time
    \ --                                       \ run-time
    COMPILE R>
    COMPILE <SELF>  COMPILE ! ;  IMMEDIATE      \ restore old <SELF>

\ To define early-bound words, use <BIND at the beginning of the colon word and
\ BIND> at the end.  In addition to putting in a BIND> prior to the semicolon,
\ you must put in a BIND> prior to any EXIT inside of the word.  Forgetting to
\ use BIND> will cause the word to crash the machine as it tries to use the old
\ <SELF> value as a return address.  If you are getting unexplained crashes, use
\ a text editor to find all of your <BIND words and manually look to see that the
\ semicolon and any EXIT words have a BIND> before them.  In general, it is best
\ to use late binding (VIRTUAL words) normally and to only use early binding if
\ speed and memory are critical and you are absolutely certain that you will
\ never want to override the member function in a derived class.

\ It is possible to write colon words which use SELF and which are called on by
\ VIRTUAL words.  These should always be made PRIVATE since they can't be called
\ by anything but VIRTUAL functions and to do so would be a bug (since they use
\ SELF and SELF isn't valid).  These aren't early-bound member functions because
\ they aren't passed an object address like the VIRTUAL functions are.  Think of
\ them as helper functions for the VIRTUAL functions.  We don't actually have
\ early-bound functions.  It is also possible to write colon words which are
\ passed one or more object addresses on the stack and which muck with the
\ object(s) in some way (by calling VIRTUAL functions and/or by modifying fields).
\ These aren't member functions, they are extraneous to the class.

\ ****** How to define fields ******

: ALIGN_INDEX  \ index -- aligned_index
    DUP NEEDED  + ;

: FIELD  \ index size -- new_index
    CREATE
        OVER ,      \ store index for use by DOES>
        +           \ return new index; for next field
    DOES>  \ object_adr -- field_adr
        @                   \ object_adr index --
        + ;                 \ field_adr --

: VIRTUAL  \ index -- new_index
    CREATE
        ALIGN_INDEX
        DUP ,       \ store index for use by DOES> (and by VIRTUAL_ADR)
        W+          \ return new index; for next field
    DOES>  \ object_adr --
        @                   \ object_adr index --
        OVER <BIND
        +                   \ field_adr --
        PERFORM             \ execute the virtual function
        BIND> ;

\ If possible, a register should be used for the variable <SELF>.  Any word that
\ accesses <SELF> should be in assembly.  All such words should be in this file.
```

```
\ I recommend against using CREATE and ;CODE to replace CREATE and DOES> since
\ it is somewhat slow this way.  I recommend using CODE to create a machine code
\ word and then executing an assembly macro which generates the proper machine
\ code and which assembles the index value into this machine code as an immediate
\ operand.  This is much faster than ;CODE which causes the pfa to be passed into
\ the machine code.  The machine code then must fetch the index value from this
\ location.  Using CODE instead of CREATE is somewhat non-traditional but, for
\ something as important as OOP, the speed increase makes it worthwhile.

: VIRTUAL_ADR  \ object_adr -- field_adr  \word: virtual_field_name
    PFA @  [ COMPILE] LITERAL  \ object_adr virtual_field_index --
    COMPILE + ;  IMMEDIATE

\ VIRTUAL_ADR is used by the constructors who need the address of the
\ virtual field within the object which they are constructing.  They need
\ this so that they can fill it in with a vector to the proper function.
\ Filling in these vectors is the primary thing that constructors do.
\ Unlike in C++, the user has to write this vector filling-in code himself.

: STRING  \ index size -- new_index     \ counted strings
    1+ FIELD ;                          \ 1+ to make room for the count byte

: INTEGER  \ index -- new_index         \ numbers
    ALIGN_INDEX  W FIELD ;

: DINTEGER  \ index -- new_index        \ double numbers
    ALIGN_INDEX  W 2* FIELD ;

: POINTER  \ index -- new_index         \ pointers to data (usually other objects)
    ALIGN_INDEX  W FIELD ;

\ STRING and FIELD provide unaligned fields of variable length.  INTEGER,
\ DINTEGER, POINTER and VIRTUAL provide aligned fields of W multiple length.


\ ****** Access to class struct ******

\ When we define a class_name, the following data go in its pfa
0 PFA_FIELD .SIZE           \ size of objects created by this class
1 PFA_FIELD .BASE           \ ptr to base class's pfa
2 PFA_FIELD .CONSTRUCTOR    \ ptr to constructor function
3 PFA_FIELD .DESTRUCTOR     \ ptr to destructor function

: SIZE_OF  \ --             \word: class_name       \ compile-time
    \ -- object_size                                 \ run-time
    PFA .SIZE @  [ COMPILE] LITERAL ;  IMMEDIATE

\ When we define an object, the following data go at its address.
0 PFA_FIELD .CLASS      \ ptr to defining class's pfa
1 PFA_FIELD .KRYSHA     \ ptr to delegating object, if there is one
2 PFA_FIELD .DATA       \ this is all of the user's FIELD and VIRTUAL data

CREATE NOTHING
    W 2* ,          \ size of the object (the .CLASS and .KRYSHA pointers)
    0 ,             \ base class pfa (the 0 is looked for by <IS_A>)
    0 ,             \ class constructor
    0 ,             \ class destructor

\ NOTHING is the base class for everything.  We build it by hand.
\ Normally classes are built with CLASS ... END_CLASS.
```

```
: SIZE   \ object_adr -- object_size
    .CLASS @        \ class_pfa --
    .SIZE @ ;

\ SIZE is primarily for incrementing a pointer through an array of objects.
\ SIZE should be used instead of SIZE_OF as much as possible.

\ ****** Constructing and destructing ******

: NADA  \ --
    ( this word does nothing) ;

\ NADA is a null-operation.  It can be given to END_CLASS as the constructor or
\ destructor of classes which don't need any specific actions here.

: ABSTRACT  \ --
    TRUE ABORT" *** You have tried to execute an abstract member function. ***" ;

\ ABSTRACT is for initializing VIRTUAL functions which are intended to be defined
\ in a derived class and which have no behavior in this class.

: <CONSTRUCT>  \ class_pfa --           \ needs SELF to be valid
    DUP .BASE @                 \ class_pfa base_class_pfa --
    DUP NOTHING = IF  DROP  ELSE  RECURSE  THEN     \ call base class constructor
    .CONSTRUCTOR PERFORM ;                          \ call our own constructor

: <DESTRUCT>  \ --                      \ needs SELF to be valid
    SELF .CLASS @  BEGIN  DUP NOTHING <> WHILE      \ class_pfa --
        DUP .DESTRUCTOR PERFORM     \ call our own destructor
        .BASE @  REPEAT DROP        \ repeat with base_class_pfa
    SELF FREE ;                     \ deallocate the memory at object_adr

: DESTRUCT  \ object_adr --
    <BIND  <DESTRUCT>  BIND> ;

: DESTRUCTS  \ first_object_adr how_many --
    DUP 1 < ABORT" *** DESTRUCTS needs a how_many parameter >= 1 ***"
    <SELF> @ >R                     \ hold old <SELF> value
    >R                              \ hold how_many value temporarily
    DUP SIZE  SWAP      \ object_size object_adr --
    R> 0 DO
        DUP <SELF> !  <DESTRUCT>    \ set <SELF> value and destroy that object
        OVER +  LOOP 2DROP
    R> <SELF> ! ;                   \ restore old <SELF> value

\ DESTRUCT and DESTRUCTS are called by the user.
\ <CONSTRUCT> and <DESTRUCT> are for internal use only.

: DELEGATE  \ object_adr field_adr --       \ to be used inside of constructors
    OVER >R  !              \ fill field_adr with object_adr
    SELF R> .KRYSHA ! ;     \ fill .KRYSHA field of object with SELF

: KRYSHA  \ -- object_adr
    SELF .KRYSHA @
    DUP [ ']  BAD_VIRTUAL = ABORT" *** .KRYSHA field was never filled in. ***" ;

\ KRYSHA is to be used inside of overridden functions of the delegatee object.
\ It provides them with the object_adr of the delegating object.
```

```
\ ****** Copying objects and testing objects' class ******

: <IS_A>  \ class_pfa target_class_pfa -- flag
    >R                                  \ hold target_class_pfa on return stack
    BEGIN  DUP WHILE    \ NOTHING's .BASE field contains a 0
        DUP R@ = IF                     \ this is it!
            R> 2DROP  TRUE EXIT THEN
        .BASE @  REPEAT                 \ repeat using base class
    R> 2DROP  FALSE ;   \ target_class_pfa not in class_pfa's inheritance chain

: IS_A  \ --                \word: class_name       \ compile-time
    \ object_adr -- flag                            \ run-time
    COMPILE @               \ class_pfa --          \ an assumed .CLASS
    PFA  [ COMPILE] LITERAL  \ class_pfa target_class_pfa --
    COMPILE <IS_A> ;
IMMEDIATE

: COULD_OCOPY  \ source_object_adr destination_object_adr -- flag
    >R                                  \ hold dst_adr on return stack
    .CLASS @  R> .CLASS @  <IS_A> ;

: OCOPY  \ source_object_adr destination_object_adr --
    >R                                  \ hold dst_adr on return stack
    DUP .CLASS @  R@ .CLASS @  <IS_A>
    0= ABORT" *** Can only OCOPY src to dst if src IS of dst's class ***"
    W+  R@ W+  R> SIZE W-  CMOVE ;   \ dst object's .CLASS field unchanged

\ OCOPY will truncate the data if src's class is derived from dst's class.
\ If they are exactly the same class, no data will be lost.
\ OCOPY adds W to the src and dst addresses and also subtracts W from
\ the size of the copy (dst's size) in order to not copy the .CLASS field.

\ It is illegal to OCOPY if src's class is a base class of dst's class since we
\ have no way of knowing what data to put in the extra fields.  It would be bad
\ programming (according to Niklaus Wirth) to initialize these extra fields to
\ some default value.  This is the hallmark of OBERON which we are emulating.
\ Don't subvert this by writing your own words to "typecast" objects; try
\ working within the constraints of OCOPY as an experiment to test Wirth's idea.

\ <IS_A> should be written in assembly to make OCOPY run quickly.
\ SIZE also to help OCOPY and because it is important on its own.


\ ****** How to define classes ******

\ All of the field defininitions are bracketed by CLASS and END_CLASS.

\ CLASS doesn't take any parameters but does require the class_name
\ and the base_class_name.  It defines the class_name as a new word.

\ END_CLASS needs to be given the vectors to the constructor and destructor
\ (as well as the data which CLASS left on the stack and which FIELD and
\ VIRTUAL have been updating).  END_CLASS fills in the class pfa fields.

: CLASS  \ -- base_class_pfa index   \word: class_name base_class_name
    CREATE
        0 , 0 , 0 , 0 ,             \ fill class_pfa with dummy values
        PFA                 \ base_class_pfa --
        DUP .SIZE @         \ base_class_pfa initial_index --
    DOES>  \ how_many -- object_adr
        <SELF> @ >R             \ hold old <SELF> value
```

```
        OVER 1 < ABORT" *** Class definers need a how_many parameter >= 1 ***"
        2DUP .SIZE @            \ how_many class_pfa how_many class_size --
        * MALLOC                \ how_many class_pfa first_object_adr --
        DUP >R
        ROT 0 DO                \ class_pfa object_adr --
            2DUP .CLASS !           \ set pointer to class_pfa in object
            DUP <SELF> !            \ set SELF for use by constructors
            OVER <CONSTRUCT>        \ call all of the constructors
            OVER .SIZE @  +   LOOP 2DROP            \ --
        R>                                          \ first_object_adr --
        R> <SELF> ! ;           \ restore old <SELF> value

: END_CLASS     \ base_class_pfa final_index ^constructor ^destructor --
                \ word: class_name
    PFA >R                  \ hold class_pfa on return stack
    R@ .DESTRUCTOR  !
    R@ .CONSTRUCTOR !
    R@ .SIZE        !       \ final_index is the object's size
    R> .BASE        ! ;

\ END_CLASS fills in the values of the class_pfa (created by CLASS)

\ We don't have any way to nest class definitions as done in C.
\ You must include a POINTER to your subclass object.  In the constructor,
\ create an instance of this class and store the object_adr in the pointer.
```

FORML, continued from page 43.

order, giving the values 1–52, with one Joker being 53 and the other 54. The key was the arrangement of the cards and the Jokers, and this had to be determined by preagreement, say a Bridge column in the newspaper. Letters were encrypted by adding the value of the letter (a–z, 1–26) to the card value, modulo 26. Then the deck was cut and shifted according to where the Jokers were located. The method is effective, simple, and inexpensive, but extremely slow (and confusing). Definitly a job his Forth program can do better.

In a third paper, Wil presented "The Most Powerful Editor That I Have Ever Used." The idea is to place a block of text into a large counted array named "Clipboard," and then to throw tiny Forth tools at it to massage the text. The tools were usually set up to process a line of text at a time, and many code examples were provided. It reminded me of Perl, and Wil conceded that you could think of it that way. ("Forth as the better Perl" — makes sense to me.)

Two papers describe using Programmable Logic Devices (PLD) to generate your own hardware. This is the opposite approach that Esson took to the problem of "disappearing hardware." John Hart, of Testra Corporation, programmed an ispLSI12032 PLD to provide the interface between an RS232 port on a PC to an RS485 security network. The device contained a baud generator, three digital filters, and a state machine. This was an example of work done with an HLDL he wrote in Forth and programmed in Forth. It generates the logic equations actually used to program the device.

Dr. C.H. Ting demonstrated his new P8 Forth processor. This is an eight-bit bus version of the P16, which has its roots in the MU21, a Forth chip that Moore and Ting developed. This project became viable with the availability of the XS40 development kit from Xess. The on-board XC4005 FPGA has the advantage of being reprogrammable. The kit also has 32K SRAM, I/O, workspace, and a parallel port for connection to a PC. By reducing the data bus to eight-bits wide, he could fit a P16 core onto the FPGA and still run a modified eForth. As it is, he used only 165 CLB logic blocks, even after adding a simple serial port, leaving 31 logic blocks for future development.

The P8 has the return and data stacks in hardware, and both are only 16 cells deep, so recursion and other stack excesses are out. It uses a long instruction word, with each 16-bit cell containing up to three five-bit instructions. Only 25 of the 32 possible instructions are implemented in hardware. Calls and jumps contain an 11-bit address, so you can only address the 2048 cells (4056 bytes) in the current page. To go beyond the page, you have to push a 16-bit address on the return stack and do a RET instruction.

The P8 project should be of interest to students and experimenters.

John Carpenter talked about "Calling Forth Methods from Java." Because Java usually takes so much time to load, he thought it best to reverse the process and have Java resident. Java can then call Forth modules in the form of Forth.DLLs. SwiftForth was used because it is Windows friendly, and he was able to demonstrate that the idea is workable.

Glen Haydon shared his thoughts on Forth Philosophy in 1999. He echoed the hope of some that the ANS Forth standard doesn't induce stasis in what has always been a dynamic language. He noted that the Internet doesn't seem to have generated the free flow of meaningful ideas on Forth, ideas that would excite newcomers to the power of Forth to improve their creativity and productivity.

Later, two papers were presented that dealt with Forth education. The attempt to reach out continues.

Dr. C.H. Ting described how he has developed a simplified and faster version of eForth that students can learn and use, and not be caught up in unnecessary details. He also outlined his Firmware Engineering Workshop, which is a four-

lesson course to teach hardware engineers the fundamentals of firmware.

Richard Haskell teaches computer science and engineering at Oakland University in Rochester, Michigan. His problem was not hardware disappearance, as much as hardware obsolescence. The assembler language course for the microchip of this year became the obsolete assembler course next year. A high-level language at least would let you concentrate on code rather than yet another language. With C/C++, however, the development environment is not the friendliest. What is needed is something small, simple, interactive, even on-board the target—something like Forth. Enter WHYP (pronounced "whip"), Words to Help You Program.

This subroutine-threaded Forth is written in C++ and assembler for the 68HC12 and 68 HC11 microcontrollers. A PC host holds the name fields with the parameter field addresses, while the parameter fields are on the target. The two are connected by a serial link. Invoking a word's name on the host will cause it to execute on the target.

All of this is explained in a course book that introduces the student to the microcontrollers and the code to control them. It leads to a discussion of the interface, the programming of simple routines, interrupts, timers, A/D conversion, fuzzy contollers, etc. All the exercises are done incrementally and interactively, allowing the student to test and program each feature of the 69HC12. By the end of the course the student understands both the 68HC12 and the usefulness of Forth.

"FORML" continues on page 54.

Mascot and Annual Award of the German Forth Interest Group

The Swap-Dragon

I really do not have the slightest idea where the Swap Dragon came from. Any suggestions? Chris Jakeman, editor of FIG UK's *Forthwrite*, thought it could have had its origin from an illustration of Leo Brodie's famous and ubiquitous book *Starting Forth*. Elizabeth Rather is reported to be quite sure that the Swap Dragon figure does indeed come from *Starting Forth* and that FORTH Inc. has given Forth-Gesellschaft specific permission to copy it solely for the purposes of a mascot.

Anyway, it has been more than ten years now that the Swap Dragon has been a mascot of Forth-Gesellschaft, the German FIG. Its foremost characteristic is the two-headedness. Can this little creature actually swap its heads? Or is it, rather, ideas which are swapped to and fro between the two heads? A symbolic representation of the notion of distributed intelligence? Decentralization. Or is it rather a manifestation of the fact that interchanging knowledge and experience is the thing most important among Forthers? I can only guess. The fact is that the Swap Dragon, in its embodied form, is a cute little bronze statue of rather heavy weight which serves the German FIG as a mascot and, at the same time, as an annual award for achievements towards Forth and merits gained in favour of Forth-Gesellschaft. It was originally imported and presented to Forth-Gesellschaft by Klaus Schleisiek, in plastic, and it recently got its current bronze form from Rolf Kretschmar, an inspired artist who, as many of us Forthers, is taking Forth as one source of his creativeness but does not insist on considering Forth as the only conceivable thing in the world.

The statue's character of also being a mascot imposes quite a duty on the respective winner of this prize: he or she has to take it in custody and see that nothing unforeseen happens to the little creature, so it can be passed on to the next in line, the next year's winner of the prize. Of course, since it would be unfair to let the same winner win the prize twice, the probability of winning the Dragon is steadily increasing from year to year. The German FIG does not have that many members, so everybody gets a fair chance. However, at the present, there might be members who will have to wait another three hundred years or so until they get the prize.

The names immortalized so far (up to 1998) in the Hall of Fame guarded by the Swap Dragon are: Michael Kalus, Heinz Schnitter, Joerg Staben, Klaus Schleisiek, Ulrike Schnitter, Jens Wilke, Joerg Plewe, Friederich Prinz, Klaus Kohl, Ulrich Hoffmann, and Bernd Paysan, in chronological order.

Yours truly, this year's custodian, does not yet know how the procedure of electing the respective new Dragon Award winner actually is evolving. He will know it next year, though, when he too will have gained the honour of being considered one of their number: the Drachenrat (Dragon Council), a Druid-like assembly of conspirators, consisting of the prize winners of the past who, at a certain time late in the evening of Forth-Tagung, the day of the Annual General Meeting of Forth-Gesellschaft, meet behind closed doors. It's always kept a great secret, well-hidden from any prospective curious intruder. Everybody would like to know more about it. Nobody has ever had the chance of learning what's really going on behind those doors, locked to the extent that even the waitress would have to swear seven oaths not to tell the public a single word of what she heard—and might have understood.

It is rumoured that there is much unintelligible muttering and mumbling from which only one word can be singled-out from time to time: *Forss*, the average native German speaker's way of pronouncing Forth, the "th" not belonging to the set of German sounds and being next to unpronounceable, for many of us. One thing, however, is for certain: the brainstorming electorial process is kept rolling by quite a number of bottles of an alcoholic kind of liquid whose exact ingredients, however, are also kept a secret. As I said before, next time I will know more about the whole procedure—and start keeping the secret from the rest of the uninitiated world.

Fred Behringer • behringe@mathematik.tu-muenchen.de

SPONSORS & BENEFACTORS

The following are corporate sponsors and individual benefactors whose generous donations are helping, beyond the basic membership levels, to further the work of *Forth Dimensions* and the Forth Interest Group. For information about participating in this program, please contact the FIG office (office@forth.org).

Corporate Sponsors

AM Research, Inc. specializes in Embedded Control applications using the language Forth. Over 75 microcontrollers are supported in three families, 8051, 6811 and 8xC16x with both hardware and software. We supply development packages, do applications and turnkey manufacturing.

Clarity Development, Inc. (http://www.clarity-dev.com) provides consulting, project management, systems integration, training, and seminars. We specialize in intranet applications of Object technologies, and also provide project auditing services aimed at venture capitalists who need to protect their investments. Many of our systems have employed compact Forth-like engines to implement run-time logic.

Computer Solutions Ltd. supplies Forth and other tools for embedded microprocessor designers and programmers in the U.K. and continental Europe. Users and developers for 18 years, COMSOL pioneered Forth under operating systems, and developed the groundbreaking chipFORTH host/target environment. Our consultancy projects range from single chip to one system with 7000 linked processors. www.computer-solutions.co.uk

Digalog Corp. (www.digalog.com) has supplied control and instrumentation hardware and software products, systems, and services for the automotive and aerospace testing industry for over 20 years. The real-time software for these products is Forth based. Digalog has offices in Ventura CA, Detroit MI, Chicago IL, Richmond VA, and Brighton UK.

Forth Engineering has collected Forth experience since 1980. We now concentrate on research and evolution of the Forth principle of programming and provide Holon, a new generation of Forth cross-development systems. Forth Engineering, Meggen/Lucerne, Switzerland – http://www.holonforth.com.

FORTH, Inc. has provided high-performance software and services for real-time applications since 1973. Today, companies in banking, aerospace, and embedded systems use our powerful Forth systems for Windows, DOS, Macs, and micro-controllers. Current developments include token-based architectures, (e.g., Open Firmware, Europay's Open Terminal Architecture), advanced cross-compilers, and industrial control systems.

The iTV Corporation is a vertically integrated computer company developing low-cost components and information appliances for the consumer marketplace. iTVc supports the Forth development community. The iTVc processor instruction set is based on Forth primitives, and most development tools, system, and application code are written in Forth.

Keycorp (www.keycorp.com.au) develops innovative hardware and software solutions for electronic transactions and banking systems, and smart cards including GSM Subscriber Identification Modules (SIMs). Keycorp is also a leading developer of multi-application smart card operating systems such as the Forth-based OSSCA and MULTOS.

www.kernelforth.com

An interactive programming environment for writing Windows NT and Windows 95 kernel mode device drivers in Forth.

MicroProcessor Engineering supplies development tools and consultancy for real-time programming on PCs and embedded systems. An emphasis on research has led to a range of modern Forth systems including ProForth for Windows, cross-compilers for a wide range of CPUs, and the portable binary system that is the basis of the Europay Open Terminal Architecture. http://www.mpeltd.demon.co.uk

RAM Technology Systems - Specialists in real-time embedded control. We develop hardware and software from initial idea to final production if required. We have developed the only commercial Forth for the PIC16Cxx range of microcontrollers and now for the AVR. If you need an embedded compiler for your new processor give us a callhttp://www.ram-tech.co.uk • irtc@ram-tech.co.uk

www.theforthsource.com

Silicon Composers (web site address www.silcomp.com) sells single-board computers using the 16-bit RXT 2000 and the 32-bit SC32 Forth chips for standalone, PC plug-in, and VME-based operation. Each SBC comes with Forth development software. Our SBCs are designed for use in embedded control, data acquisition, and computation-intense control applications.

T-Recursive Technology specializes in contract development of hardware and software for embedded microprocessor systems. From concept, through hardware design, prototyping, and software implementation, "doing more with less" is our goal. We also develop tools for the embedded marketplace and, on occasion, special-purpose software where "small" and "fast" are crucial.

Tateno Dennou, Inc. was founded in 1989, and is located in Omecity Tokyo. Our business is consulting, developing, and reselling products by importing from the U.S.A. Our main field is DSP and high-speed digital.

ASO Bldg., 5-955 Baigo, Ome,Tokyo 198-0063 Japan
+81-428-77-7000 • Fax: +81-428-77-7002
http://www.dsp-tdi.com • E-mail: sales@dsp-tdi.com

Taygeta Scientific Incorporated specializes in scientific software: data analysis, distributed and parallel software design, and signal processing. TSI also has expertise in embedded systems, TCP/IP protocols and custom applications, WWW and FTP services, and robotics. Taygeta Scientific Incoporated • 1340 Munras Avenue, Suite 314 • Monterey, CA 93940 • 831-641-0645, fax 831-641-0647 • http://www.taygeta.com

Triangle Digital Services Ltd.—Manufacturer of Industrial Embedded Forth Computers, we offer solutions to low-power, portable data logging, CAN and control applications. Optimised performance, yet ever-increasing functionality of our 16-bit TDS2020 computer and add-on boards offer versatility. Exceptional hardware and software support to developers make us the choice of the professional.

Individual Benefactors

Makoto Akaishi
Everett F. Carter, Jr.
Edward W. Falat
Michael Frain
Guy Grotke
Bjorn Gruenwald
John D. Hall
Guy Kelly
Zvie Liberman
Marty McGowan
Andrew McKewan
Peter Midnight
John Muller
Gary S. Nemeth
Marlin Ouverson
John Phillips
Thomas A. Scally
Martin Shann
Werner Thie
Richard C. Wagner