$10 Volume XVIII, Number 6 March 1997 April

FORTH DIMENSIONS

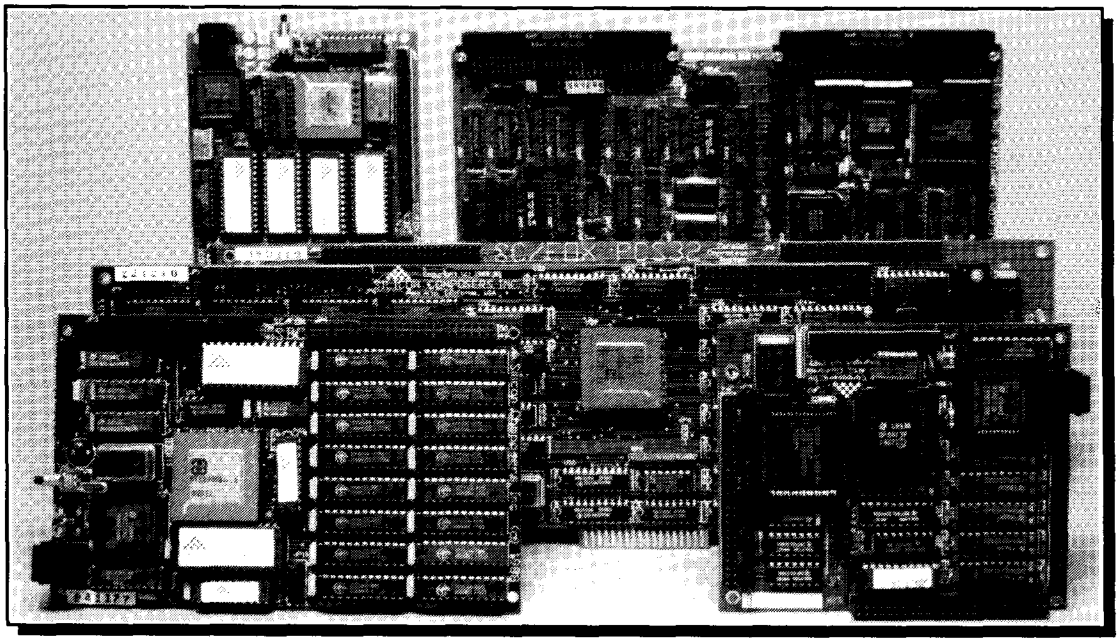
Fuzzy Forth

A Modest Proposal

Object-Oriented ANS Forth

Forth Coding Style

Digital Filters



Contents

Features

Editorial

Coding conventions is one of those topics that catalyzes contrasting opinions and justifications in a roomful of Forth programmers, any two of whom will have at least three different ways of looking at it. Everyone agrees it is a good idea but no one would dare enforce it via compiler requirements, which would be something like trying to elevate civil law to the status of natural law.

Then there's the difference between what we say in public, what we believe about ourselves, and what we practice. For a reminder (and a test), dig out some unmemorable code you wrote more than a year ago. What do you make of it? What would you make of it in five years? Would a stranger find in it not only understanding, but also a good example or two by which to improve his own code? Is *any* of it reusable?

Of course, coding conventions assume their greatest importance, if not their only one, for programming teams. With the relative freedom of style and technique that Forth in particular provides, an individual's code can become as, well, *individual* and as recognizable as his hairstyle or her posture. In the extreme this can, of course, lead to personality-based code conflicts in which two routines simply will not function correctly in the same program together. Then come, in due course, the arbitration skills of an integrator (possibly even a small team) with a bag full of tricks and experience; or a manager with seemingly arbitrary decisions which, elegant and reasonable or not, keep the project moving forward.

Sometimes the greatest success of the code integrator is getting the team members to share a new way of looking at the project, or to agree at a philosophical level on the merit of a Forth technique or tradeoff, or simply to back off a bit and remember the big picture. Then, code-level decisions spring from a common mindset and clashes are avoided entirely, or at least are defused sooner, before becoming entrenched in opposition.

Under a production schedule, every bit of this that arises is felt as a blow to the budget and timeline, and as more internal personal pressure. It definitely hinders one's pleasure in the job and may even lead to serious functional defects and the dreaded last-minute surprise.

So the voluntary price we pay—for the posterity of our code and the longevity of our jobs and to generally play nicely and get along with others—is to adopt a set of coding guidelines and follow it closely. It becomes part of the personal aesthetic by which we judge the quality of our code; and because we know how real life works on the job, we don't treat these conventions as cosmetics to be applied as a last iteration in the development process.

In the end, all this becomes second nature—your code can still be distinctly your own, but it will fall within a carefully defined intersection of functional compatibility, general readability, and style. Like handwriting. (You could, in fact, write gibberish and still adhere to the conventions, but that doesn't argue against their suitability.)

Coding conventions are a social construct that, like traffic laws, everyone with a little urging can be convinced is a good thing. It's in the details that it gets thorny, so we don't have a generally endorsed document that demonstrates good style. We have a few words here and there. A rare paper or article. A short discussion about phrasing and indentation and whether to give the semi-colon a line of its own. How many hyphens in your stack diagrams?

Steve Pelc of England's MPE has generously contributed his company's official, documented coding standard for publication in *Forth Dimensions*. We will see over time (because space permits us to publish it only serially) that such a document must have a breadth of philosophy encoded in its details. Top-down design, bottom-up writing.

And there are many details. This demonstrates, in an admittedly non-glamorous way, the amount of conventional wisdom there is about how to write Forth. Not everyone will agree with the way the particulars are handled, but most will agree that (a) the particulars are important, and (b) a team of any size that is going to work together successfully on a project of any scope will have to adopt some kind of formatting guide. *MPE Forth Layout Standard* version 4.00.003 (2 July 1996) is one such that has evolved over time and which exemplifies the important issues and how this company's management and staff addresses them.

—Marlin Ouverson
editor@forth.org

Forth Dimensions
Volume XVIII, Number 6
March 1997 April

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: 408-37-FORTH Fax: 408-373-2845

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 the Forth Interest Group, 100 Dolores Street, suite 183, Carmel, California 93923. Second-class postage paid at Seaside, CA. POSTMASTER: Send address changes to *Forth Dimensions*, 100 Dolores Street, suite 183, Carmel, California 93923."

Office News

Contents

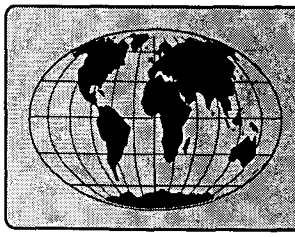
It's been an exciting start to 1997 here at the FIG main office. As I write, we're surrounded by the boxes of inventory we hauled from Oakland on the first weekend of February. With the help of our new office assistant, Julie Stone, we're fast on the track to getting it all organized. We've been able to fill orders for product from this office, and the next issue of *Forth Dimensions* will be bundled and sorted and shipped from here. Wish us luck!

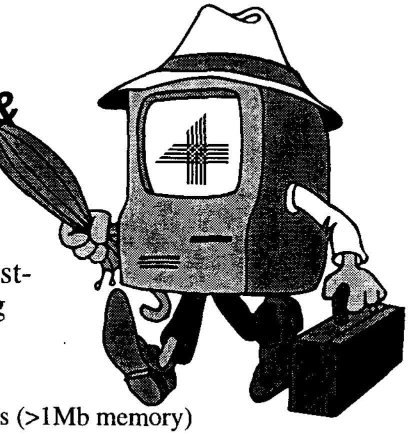
Our office hours have changed a bit; you can usually find Julie or myself here from 9:00 a.m. – 1:30 p.m. Don't be fooled though—as you probably realize, many hours are put in during the afternoons, evenings, and on the weekend (as some of you who have called during those times have found out). However, we try to personally answer the phones during the hours noted above; at other times, you may get the answering machine. Please leave your message, we will get back to you. I want to thank you again all for your patience and good humor as we become more efficient in processing your needs.

During the first six weeks of 1997, we had 47 new-member requests from the Internet. 25 of those requests originated outside of the U.S., from countries like Malaysia, Hong Kong, the Ukraine, Russia, Brazil, Germany, Singapore, South Korea, the Netherlands, Indonesia, and Sweden. The Forth Interest Group is truly international!

In the coming months, we hope to be able to provide support for those of you who are interested in re-starting a FIG Chapter. Let us know what you need, we will try to provide it. If you've got time or energy or interest to volunteer, let us know and during this coming year we will use you. Thanks again for being such a great group, it makes all the work feel worthwhile.

Trace Carter
Forth Interest Group, Administrative & Sales Office
100 Dolores Street, Suite 183
Carmel, California 93923 U.S.A.
408-373-6784 (voice)
408-373-2845 (fax)
office@forth.org (e-mail)





Adapted from a paper presented at the annual Rochester Forth Conference

Fuzzy Forth

An ANS-Compliant Extension

Rick VanNorman
Manhattan Beach, California

Fuzzy logic is still a popular topic in the computer and electronics field. The controversy surrounding it does not detract from its advantages as an implementation technique. Many products are sold which provide programmers with fuzzy logic interfaces for their programs, but rarely do these packages come with source code which can be examined and understood by the programmer.

This paper presents an overview of fuzzy logic, a set of ANS-compliant source code extensions to Forth to implement a fuzzy logic inference engine, and a simple example of a Fuzzy Forth control program.

Introduction

Most programming languages in use today deal with issues of true and false, on and off, black and white. Fuzzy logic is a technique for allowing a program on a digital computer to deal with the shades of gray found in most real-world situations.

Fuzzy logic, by itself, is simply the realization that an apple with one bite taken out of it is still at least 90% of an apple, and that it doesn't completely cease to be an apple until it is completely gone. Fuzzy logic, as applied to computer programming is a realization of this logic in a series of discrete steps: *input* (or fuzzification), *rule evaluation*, and *conclusions* (or de-fuzzification). Fuzzy Forth is an ANS-compliant toolkit for Forth which allows the user to easily define sets for input and fuzzification of crisp or measured values, rules for reasoning about the inputs, and a simple specification of the output ranges for converting the results of the rules into usable control values. Fuzzy Forth defines two basic data structures, one each for input and output.

Input (Fuzzification)

Fuzzification is the act of classifying a measured value (called a *crisp*) into a set of fuzzy values. For example, a real-world temperature of 68 degrees could be classified as (cold=10% true), (warm=70% true), and (hot=0% true). These values represent the fuzzy memberships of the crisp value on the fuzzy variable. Each fuzzy variable has a lower bound, below which the membership is zero; an upper bound, above which it is zero; and a body in which the membership varies from 0% to 100%. This body is normally shaped like a trapezoid or a triangle. In a system of multiple inputs, each input would have its own set of fuzzy membership functions, each requiring evaluation at each iteration of the program.

The input data structure consists of the input, or crisp variable, and a set of membership sets which span the range of the input.

Input values are evaluated for membership in each of the sets in the input's linked list, and each membership is assigned a *fuzzy value* in the range (0–256) which corresponds to the nominal range of 0–100%. This is called "fuzzification."

Each membership set has a body defined by four points:

`lm` The left-most point of the body. Every value less than this assumes the fuzzy value of 0.

`lt` The left-top corner of the body. Values which are between `LM` and `LT` are assigned a fuzzy value based on the slope of the line between `LM` and `LT`.

`rt` The right-top corner of the body. Values between `LT` and `RT` are assigned the fuzzy value of 100% (256).

`rm` The right-most point of the body. All values greater than this are given the fuzzy value 0.

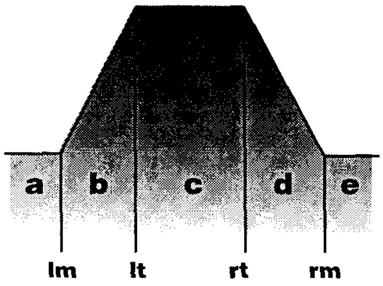
The words `INPUT` and `MEMBER` define and build the structure shown in Figure One.

`APPLY`ing a crisp value to an `INPUT` generates a set of fuzzy `MEMBER`ship values. The functional use of an input variable is:

```
( n) DERROR APPLY
```

which takes the value *n* and calculates its fuzzy value in each of the membership sets of the fuzzy input `DERROR`. An input must be evaluated via `APPLY` before any rules can be executed. When an `INPUT` is invoked by name, it returns the address of the data structure where its crisp input value is stored. When a `MEMBER` is executed, it returns the value calculated by `APPLY` for the crisp value in its parent `INPUT`.

Figure One. The input function and its data structures.

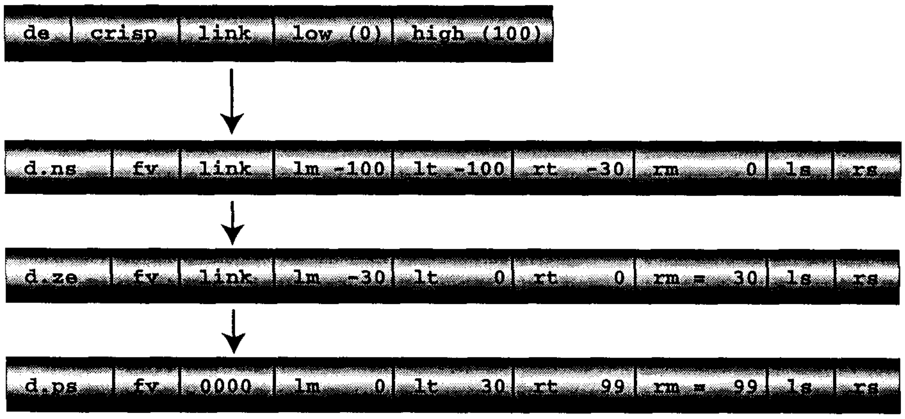


A typical membership function.
Note that points LT and RT can be equal, resulting in a triangle, and that LM:LT and RT:RM could be equal, resulting in a step function.

LM <= LT <= RT <= RM

```
-100 100 INPUT DERROR
     -100 -100   -30    0 MEMBER D.NS
      -30    0     0   30 MEMBER D.ZE
        0   30    99   99 MEMBER D.PS
```

A typical input function,
with a diagram of the structure that words `INPUT` and `MEMBER` build. The member `D.ZE` forms a triangle, the members `D.PS` and `D.NS` form trapeziods, each with a single vertical edge



This structure makes rule building and evaluation simpler.

Rule Evaluation

The rulebase is the mechanism that ties `INPUTS` to `OUTPUTS`. Forth simplifies rule specification because it allows the definition of a language extension which can deal with fuzzy logic in a user-defined manner instead of trying to create a set of fuzzy logic tools in an existing syntax.

A *rule* is a statement of condition with at least one input and one consequence. Multiple inputs may be combined with fuzzy operators which resemble conventional logic, such as `AND` or `OR`, and may be negated with `NOT`. Rules are of the form

```
xyz abc & => ddd
```

which says "take the minimum of the membership functions as evaluated by the `MEMBER` words `XYZ` and `ABC`, leaving the result on the stack to be absorbed by the `SINGLETON` definition `DDD` (a fuzzy output accumulator). Valid operations are `AND` (`&`), `OR` (`|`), and `NOT` (`~`), with each "logical" value in the range 0–256. The chosen convention for fuzzy logical operators in Fuzzy Forth is shown in Figure Three.

Rule evaluation is simply logic applied to the fuzzy variables. For example, one rule might be "if the temperature is cold and the humidity is high, then the fan-speed is fast." The act of evaluating this rule requires that we determine the fuzzy value of the temperature in the membership function "cold" and combine it with the fuzzy value of the humidity in the membership function "high." Since each of these memberships has a strict numeric value which may range from zero to 100%, we need to combine them with the fuzzy operator AND which is typically implemented as the minimum of the two values. This results in answers that carry the lowest common value, or *weight*, of the inputs. So, if `TEMP.COLD` is 10% true and `HUMIDITY.HIGH` is 40% true, the result of ANDing them together is 10% true. This result is typically accumulated into a fuzzy output variable, e.g., keeping the overall maximum value of all the fan-speed rule evaluations.

The generalization of this rule evaluation method might be summed up as: for each rule, take the smallest input, because this is the limit of how much this rule matters; then, remember only the largest effect to each output, since if one rule evaluated to more than another, it must have more influence on the output. This technique is called the *min-max rule* and is only one of many rule evaluation schemes proposed by the fuzzy logic theorists.

Rulebases are typically built from FAMs (Fuzzy Associative Memories). A rulebase in Fuzzy Forth shown in Figure Four.

Conclusions (De-Fuzzification)

After fuzzifying the inputs and evaluating the rules, we have accumulated a fuzzy truth value for each of the output membership functions. This is equivalent to the fuzzy input memberships that we calculated from the crisp inputs. This fuzzy truth value is then converted to a crisp value for output by the word `CONCLUDE`. Its use is:

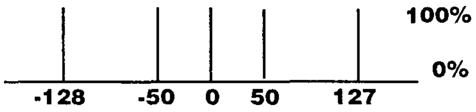
```
SPEED CONCLUDE ( n)
```

which traverses the list of `SINGLETON`s associated with `SPEED`, calculating the center of mass of the function described by their fuzzy values and weights.

For example, we might have concluded that `FAN_SPEED.SLOW` was 40% true and `FAN_SPEED.FAST`

Figure Two. The output function and its data structures.

A typical output function, shown as a set of singletons on the number line. Each singleton has a weight of 1, which is scaled by rule evaluation.



```
OUTPUT SPEED

   -128 SINGLETON S.NM
    -50 SINGLETON S.NS
      0 SINGLETON S.ZE
     50 SINGLETON S.PS
    127 SINGLETON S.PM
```

The OUTPUT defines the **root** of the structure, and each SINGLETON adds a **node** to the linked list.

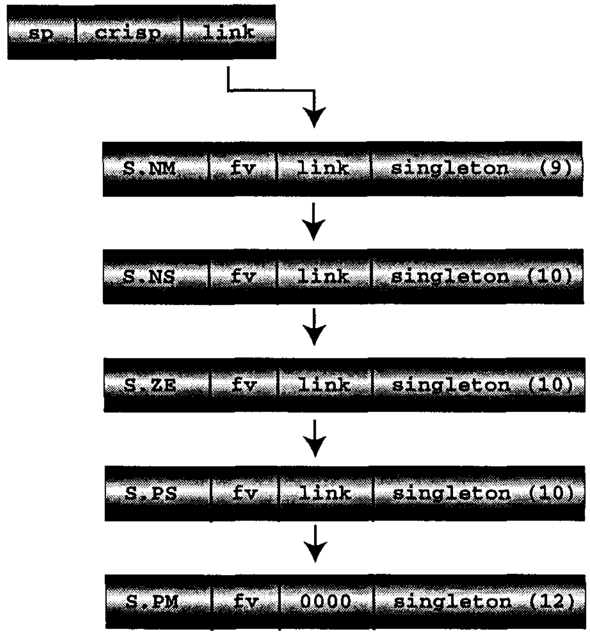


Figure Three. Logical operators in Fuzzy Forth.

&	*fuz1 fuz2 --fuz3*	Fuzzy logic AND. Returns the minimum of the two inputs.
|	*fuz1 fuz2 --fuz3*	Fuzzy Logic OR. Returns the maximum of the two inputs.
~	*fuz1 --fuz2*	Fuzzy logic NOT. Returns the eight-bit compliment of the input.
=>	*fuz1 --fuz1*	Syntactic sugar, read as "implies."

A rule such as `E.NM D.NS & => S.PM` is read, "A negative medium error AND a negative small delta-error implies a small positive motor speed."

Figure Four. A Fuzzy Associative Memory rulebase for a thermostat.

A rulebase for describing a fuzzy controller which generates a correction in speed. Inputs are ERROR and DERROR, output is SPEED.

	derror		
error	**ns**	**ze**	**ps**
nm	pm	pm	ps
ns	pm	ps	ze
ze	ps	ze	ns
ps	ze	nm	nm
pm	ns	nm	nm

nm	=	*negative medium*
ns	=	*negative small*
ze	=	*zero*
ps	=	*positive small*
pm	=	*positive medium*

was 20% true, but this does not tell us how fast to run the fan. To make the fuzzy outputs useful, we must evaluate each fuzzy accumulator and calculate a crisp value for each output.

This is done by taking the center of mass of the output functions, as weighted by their fuzzy truth values. There are many different methods, each with its own merits, of recombining the fuzzy outputs into crisp values. The technique used in Fuzzy Forth is singletons, or *unit masses*, chosen for its simplicity of implementation and speed of execution. The next most common technique uses *area masses*, which allows for a wider output function and results in somewhat smoother results (but is computationally slower). Kosko and others have shown that the singleton is adequate to represent virtually all output functions by simply placing the unit masses closer together or further apart to achieve the overall output mass density desired.

The structure of an OUTPUT is similar to that of an INPUT, consisting of a root, which is the output variable, and a linked list of SINGLETONS. (See Figure Two.) A SINGLETON is the "fuzzy accumulator," storing the maximum value of any rule of which it was the target. This value is considered the truth value of the SINGLETON and is used to scale the weight of its fixed mass. (By default, SINGLETONs have a mass of *one;* extensions to Fuzzy Forth could allow for other masses, resulting in smoother output functions.) The value of the OUTPUT is determined after all the rules have fired by calculating the center of mass of its scaled SINGLETONs.

To generate a conclusion, we accumulate a *sum of area* (SOA) and a *sum of products* (SOP) term for each set of singletons (which are taken to be point sources whose weight was accumulated by the rule evaluations). Dividing

SOP by SOA results in the center of mass of the point sources, which we take to be the de-fuzzified value of the output variable.

Summary

In summary, Forth can be extended to incorporate a fuzzy logic inference engine, with memory- and speed-efficient data structures. The resulting language provides a simple and natural interface for building applications using fuzzy logic.

Fuzzy logic has been used in various real-world scenarios where the input parameters vary greatly, and a continuously varying output is desired. Examples of this include: deciding the shift points for automatic transmissions, elevator demand scheduling, thermostats, and pattern recognition. Many control problems which are difficult in standard control methodology are much more easily addressed by fuzzy logic. For instance, the favorite "toy" problem of fuzzy logic theorists is balancing an upright pole on a two-axis controller. This is akin to balancing an upright broom on your fingertip. Fuzzy Logic can not only balance a pole, but some hearty researchers have even demonstrated a segmented-pole-balancing fuzzy controller.

Fuzzy logic is not an answer to every problem, but can simplify the implementation of some control problems by allowing a degree of imprecision in their specification. Fuzzy Forth provides a toolkit for the use of fuzzy logic in Forth.

References

Jamshidi, M., N. Vadiee, T. J. Ross, editors. *Fuzzy Logic and Control.* New Jersey: PTR Prentice-Hall, 1993.

Kosko, B. *Fuzzy Thinking: The New Science of Fuzzy Logic.* New York: Hyperion, 1993.

Kosko, B. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence.* New Jersey: Prentice-Hall, 1992.

Rick VanNorman has been programming in Forth professionally since 1979, and currently works for FORTH, Inc. He earned a degree in computer science in 1982, and has worked with Forth in industrial control, manufacturing, "star wars" research, and electronic payment systems. Rick was a member of the Harris RTX team and was part of the infamous "Binar" design group. He can be reached by e-mail at the rick@thunder-ink.com address.

Listing One. The Fuzzy Forth language extension.

```
\ Fuzzy Forth, (C) 1993, 1994, 1997 Rick VanNorman

\ store two entries in the dictionary
: 2, ( n1 n2 -- )
   SWAP , , ;

\ division by zero is allowed, and results in a zero returned
: DIV ( n1 n2 -- n3 )
   dup if / else nip then ;

\ calculate the slope between x1=p2 and x2=p1, assuming ONE = 65536 and that
\ the y1=0 and y2=ONE.  if x1=x2, the slope is properly infinity, but
\ we only apply the calculated slope where we know there _is_ a slope.
\ FUZZIFY will trap this condition.

: SLOPE ( P2 P1 -- N )
   -  65536. ROT ?DUP IF  UM/MOD NIP ABS  ELSE  2DROP 0  THEN ;

VARIABLE MLINK       \ Temp value for member links

: LINK,              \ store a link in the current definition
   HERE MLINK @ !  HERE MLINK !  0 , ;

\ -----
\ input variable and membership function definition

\ name a "crisp" input, specifying its range
: INPUT ( low high "name" -- )
   CREATE
      0 ,                                  \ Crisp storage
      HERE MLINK !  0 ,                    \ Initial link
      2, ;                                 \ the low and high ranges

\ Set the slope of the membership function being defined. The slope
\ is calculated at compile time for run-time efficiency.
```

```
: SET_SLOPE ( ilink -- )
   >R  R@ 1 CELLS + 2@ SLOPE  R@ 5 CELLS + !
       R@ 3 CELLS + 2@ SLOPE  R> 6 CELLS + ! ;

\ Name a membership function associated with the current crisp input.
\ The values {lm lt rt rm} define the curve under which the function
\ is active.
: MEMBER ( lm lt rt rm "name" -- )
   CREATE
      0 ,  LINK,                          \ fuzzy value and link to prev
      2>R 2, 2R> 2,   0 0 2,              \ store points, left to right
      MLINK @ SET_SLOPE                   \ calculate slopes of the edges
   DOES>  @ ;                             \ return the fuzzy value

\ -----
\ output variable and de-fuzzification set definition

\ Name a fuzzy output function.  Outputs are associated with a
\ list of SINGLETON fuzzy accumulators.
: OUTPUT
   CREATE
      0 ,                                    \ CRISP STORAGE
      HERE MLINK !  0 , ;                    \ INITIAL LINK

\ Name a singleton and associate it with the output function being
\ defined. The value of the singleton is its ordinal position based
\ solely on the output range desired.
: SINGLETON ( value "name" -- )
   CREATE
      0 , LINK,                              \ fuzzy value and link to prev
      ( value) ,                             \ and the singleton value itself
   DOES>  DUP @ ROT MAX SWAP ! ;             \ accumulate rule evaluations

\ initialize an output set

: CLEAR_OUTPUT ( addr -- )
   CELL+ BEGIN                               \
      @ ?DUP WHILE  0 OVER CELL- !           \ following the links, write 0s
   REPEAT ;                                  \ to the fuzzy accumulators

\ RULE EVALUATION
\ -----

: &   MIN ;
: |   MAX ;
: ~   256 SWAP - ;

: => ; IMMEDIATE

\ conclusions after rule evaluation
\ -----

VARIABLE SOA      \ sum of area
VARIABLE SOP      \ sum of products

: CONCLUDE ( OVAR -- CRISP )
```

```
   0 SOA !  0 SOP !              \ CLEAR SUM OF AREA, SUM OF PRODUCT TERMS
   DUP >R                        \ save the address of the output var
   CELL+ BEGIN                   \ and pointing to the singleton's link
      @ ?DUP WHILE               \ repeat while not at the end of the list
      DUP CELL- @  DUP SOA +!    \ SOA is the sum of all fuzzy values
      OVER CELL+ @               \ SOP is sum of all (fuzzy * weight) values
      * SOP +!                   \
   REPEAT                        \
   SOP @ SOA @ DIV               \ divide SOP by SOA, giving the center of mass
   DUP R> ! ;                    \ which we interpret as the de-fuzzification

\ fuzzify the inputs
\ -----
\
\ we fuzzify the inputs by applying the crisp value to each of the
\ membership functions and evaluating its position in the membership.
\

\ fuzzyfiy the crisp value on the left edge of the membership (ie region B)
\ via the slope of that line segment and its X position relative to the
\ start of the line segment.

: FUZLEFT ( CRISP MEMBER -- FUZZY )
   DUP >R  @ -  R> 4 CELLS + @ 256 */  ABS ;

\ fuzzyfiy the crisp value on the right edge of the membership (ie region D)
\ via the slope of that line segment and its X position relative to the
\ start of the line segment.

: FUZRIGHT ( CRISP MEMBER -- FUZZY )
   DUP >R  3 CELLS + @ SWAP -  R> 5 CELLS + @ 256 */  ABS ;

\ given a crisp number, a membership function, and a cell offset, return
\ the crisp value and the range point for comparison

: IN ( crisp member n -- crisp member crisp range )
   >R 2DUP R> CELLS + @ ;

\ given a crisp value and a pointer to the membership points, figure out
\ where the crisp value falls (left to right) and assign a membership
\ value for it.  membership values are in the range (0--256)

: FUZZIFY ( CRISP MEMBER -- FUZZY )
   0 IN  <  IF ( BELOW LEFTMOST )  2DROP 0    EXIT THEN   ( is zero)
   1 IN  <  IF ( LM <= C <= LT  )  FUZLEFT    EXIT THEN   ( is calculated)
   2 IN  <= IF ( LT <= C <= RT  )  2DROP 256  EXIT THEN   ( is "one")
   3 IN  <= IF ( RT <= C <= RM  )  FUZRIGHT   EXIT THEN   ( is calculated)
               ( ABOVE RIGHTMOST)  2DROP 0     ;          ( is zero)

\ evaluate the crisp value in the specified membership function.  the
\ result is stored in the first cell.  the address of the member passed
\ to this routine is the a pointer to its link field.

: APPLIES ( crisp member -- )
   TUCK  CELL+ FUZZIFY  SWAP CELL- ! ;
```

```
\ apply the crisp input to the specified fuzzy input variable

: APPLY ( crisp input -- )
   2DUP !  CELL+ BEGIN              \ keep the original crisp value
      @ ?DUP WHILE                  \ and scan through each of the memberships
      ( CRISP LINK)  2DUP APPLIES   \ applying the crisp to them
   REPEAT DROP ;
```

Listing Two. An example in Fuzzy Forth which implements a thermostat.

```
\ A simple Fuzzy Forth example
\ -----
\ Adapted for Fuzzy Forth by Rick VanNorman 1993,1994
\
\ taken from
\ _A Fuzzy Two-Axis Mirror Controller for Laser Beam Alignment_
\     by: Richard D. Marchbanks
\ published in
\ _Fuzzy Logic and Control_
\     edited by: Mohammad Jamshidi, Nader Vadiee, and Timothy J. Ross
\     1993, PTR Prentice Hall, Englewood Cliffs, New Jersey
\     ISBN 0-13-334251-4
\

\ the input memberships
\ -----

-128 127 INPUT ERROR

      -128 -128 -100 -20 MEMBER E.NM
      -128  -20  -20  -3 MEMBER E.NS
       -20    0    0  20 MEMBER E.ZE
         3   20   20 127 MEMBER E.PS
        20  100  127 127 MEMBER E.PM

-100 100 INPUT DERROR

      -100 -100  -30   0 MEMBER D.NS
       -30    0    0  30 MEMBER D.ZE
         0   30   99  99 MEMBER D.PS

\ the output function, as singletons
\ -----

OUTPUT SPEED

   -128 SINGLETON S.NM
    -50 SINGLETON S.NS
      0 SINGLETON S.ZE
     50 SINGLETON S.PS
    127 SINGLETON S.PM

\ the rulebase
\ -----
\ the rulebase for generating a correction in speed, based on a
\ simple FAM for inputs ERROR and DERROR, output is SPEED
\
\        |error
\        |  nm    ns    ze    ps    pm
\ derror |-----------------------------
\     ns |  pm    pm    ps    ze    ns
\     ze |  pm    ps    ze    ns    nm
\     ps |  ps    ze    ns    nm    nm
\

: RULES
   SPEED CLEAR_OUTPUT
      E.NM D.NS & => S.PM    E.NM D.ZE & => S.PM    E.NM D.PS & => S.PS
      E.NS D.NS & => S.PM    E.NS D.ZE & => S.PS    E.NS D.PS & => S.ZE
      E.ZE D.NS & => S.PS    E.ZE D.ZE & => S.ZE    E.ZE D.PS & => S.NS
      E.PS D.NS & => S.ZE    E.PS D.ZE & => S.NM    E.PS D.PS & => S.NM
      E.PM D.NS & => S.NS    E.PM D.ZE & => S.NM    E.PM D.PS & => S.NM
   ;

\ a sample driver to use it all...
\ -----

VARIABLE OLD_ERROR

DEFER INPUT_ERROR ( -- error )
DEFER MOTOR       ( speed -- )

: ERROR_TERMS ( new -- new delta )
   DUP OLD_ERROR @ OVER OLD_ERROR !  - ;    \ error derror

: CONTROL ( new_error delta_error -- speed )
   ERROR_TERMS       \ leave ERROR and DERROR on stack
   DERROR APPLY      \ fuzzyify derror
   ERROR APPLY       \ fuzzyfiy error
   RULES             \ execute the rulebase
   SPEED CONCLUDE ;  \ determine what speed to apply to achieve target

: TEST
   BEGIN
      INPUT_ERROR ERROR_TERMS CONTROL MOTOR
      KEY?
   UNTIL ;

\ produce a surface array, suitable for framing, of the fuzzy function
: SURFACE
   128 -128 DO             \ loop over the error range
      I ERROR APPLY        \ fuzzify the error term
      CR                   \
      100 -100 DO          \ loop over the derror range
         I DERROR APPLY    \ and fuzzify the delta error
         RULES             \ run the rules
         SPEED CONCLUDE    \ generate the output
         5 .R              \ and print
      10 +LOOP             \ limit to 20 terms
   10 +LOOP ;              \
```

Object-Oriented Programming in ANS Forth

Andrew McKewan
Austin, Texas

This article describes the use and implementation of an object-oriented extension to Forth. The extension follows the syntax in Yerk and Mops, but is implemented in ANS Forth.

Why I Am Doing This

When I first began programming in Forth for Windows NT, I became aware of the huge amount of complexity in the environment. In looking for a way to tame this complexity, I studied the object-oriented Forth design in Yerk. Yerk is the Macintosh Forth system that was formerly marketed as a commercial product under the name Neon. It implemented an environment that allowed you to write object-oriented programs for the Macintosh.

While much of Yerk was Macintosh-specific, the underlying class/object/message ideas were quite general. I ported these to Win32Forth, a public-domain Forth system for Windows NT and Windows 95. However, in both Yerk and Win32Forth, much of the core system is written in assembly language and is very machine-specific. Additionally, both systems modified the outer interpreter to adapt to the new syntax.

What I hope to accomplish here is to provide any ANS Forth system with the ability to use the object-oriented syntax and programming style in these platform-specific systems. In doing so, I have sacrificed some performance and a few of the features.

Object-Oriented Concepts

The object-oriented model closely follows Smalltalk. I will first describe the terminology used in this model: objects, classes, messages, methods, selectors, instance variables, and inheritance.

Objects are the entities used to build programs. Objects contain private data that is not accessible from outside the object. The only way to communicate with an object is by sending it a message.

A *message* consists of a selector (a name) and arguments. When an object receives the message, it executes a corresponding method. The arguments and results of this method are passed on the Forth stack.

A *class* is a template for creating objects. Classes describe the instance variables and methods for the object. Once a class is defined, you can make many objects from that class. Each object has its own copy of the instance variables, but shares the method code.

Instance variables are the private data belonging to an object. Instance variables can be accessed in the methods of the object, but are not visible outside the object. Instance variables are themselves objects with their own private data and public methods.

Methods are the code that is executed in response to a message. They are similar to normal colon definitions, but use a special syntax using the words `:M` and `;M`. You can put any Forth code inside a method, including sending messages to other objects.

Inheritance allows you to define a class as a subclass of another class, called the superclass. This new class "inherits" all the instance variables and methods from the superclass. You can then add instance variables and methods to the new class. This can greatly decrease the amount of code you have to write, if you design the class hierarchy carefully.

I hope to provide object-oriented syntax and programming style to any ANS Forth system...

How to Define a Class

The example of a `Point` class illustrates the basic syntax used to define a class (see Figure One).

The class `Point` inherits from the class `Object`. `Object` is the root of all classes and defines some common behavior (such as getting the address of an object or getting its class) but does not have any instance variables. All classes must inherit from a superclass.

Next we define two instance variables, `x` and `y`. Both of these are instances of class `Var`. `Var` is a basic, cell-sized class similar to a Forth variable. It has the methods `Get:` and `Put:` to fetch and store its data.

The `Get:` and `Put:` methods of class `Point` access its data as a pair of integers. They are implemented by

Figure One. Example of a `Point` class.

```
:Class Point  <Super Object

  Var x
  Var y

  :M Get:  ( -- x y )  Get: x  Get: y  ;M
  :M Put:  ( x y -- )  Put: y  Put: x  ;M

  :M Print:  ( -- )
     Get: self  SWAP  ." x = " .   ." y = " .  ;M

:M ClassInit:   1 Put: x  2 Put: y  ;M

;Class
```

sending `Get:` and `Put:` messages to the instance variables. `Print:` prints out the *x* and *y* coordinates.

`ClassInit:` is a special initialization method. Whenever an object is created, the system sends it a `ClassInit:` message. This allows the object to perform any initialization functions. Here we initialize the variables *x* and *y* to a preset value. Whenever a point is created, it will be initialized to these values. This is similar to a constructor in C++.

Not all classes need a `ClassInit:` method. If a class does not define the `ClassInit:` method, it will inherit the one in class `Object` that does nothing.

Creating an Instance of a Class

Now that we have defined the `Point` class, let's create a point:

```
Point myPoint
```

As you can see, `Point` is a defining word. It creates a Forth definition called `myPoint`. Let's see what it contains:

```
Print: myPoint
```

This should print the text *x = 1 y = 2* on the screen. You can see that the new point has been initialized with the `ClassInit:` message.

Now we can modify `myPoint` and we should see the new value:

```
3 4 Put: myPoint
Print: myPoint
```

Notice that, in the definition of `Point`, we created two instance variables of class `Var`. The object-defining words are "class smart" and will create instance variables if used inside a class, and global objects if used outside a class.

Sending a Message to Yourself

In the definition of `Print:` we used the phrase `Get: self`. Here we are sending the `Get:` message to ourselves. `Self` is a name that refers to the current object. The compiler will compile a call to `Point`'s `Get:` method. Similarly, we could have defined `ClassInit:` like this:

```
:M ClassInit:
1 2 Put: self  ;M
```

This is a common factoring technique in Forth and is equally applicable here.

Creating a Subclass

Let's say we wanted an object like `myPoint`, but which printed itself in a different format.

```
:Class NewPoint  <Super Point

    :M Print:  ( -- )
       Get: self  SWAP  0 .R
       ." @" .  ;M

;Class
```

A subclass inherits all the instance variables of its superclass, and can add new instance variables and methods of its own, or override methods defined in the superclass. Now let's try it out:

```
NewPoint myNewPoint

Print: myNewPoint
```

This will print *1@2* which is the Smalltalk way of printing points. We have changed the `Print:` method, but have inherited all the other behavior of a `Point`.

Sending a Message to Your Superclass

In some cases, we do not want to replace a method but just add something to it. Here's a class that always prints its value on a new line:

```
:Class CrPoint  <Super NewPoint

    :M Print:  ( -- )
       CR  Print: super  ;M

;Class

CrPoint myCrPoint
Print: myCrPoint
```

When we use the phrase `Print: super` we are telling the compiler to send the print message that was defined in our superclass.

Indexed Instance Variables

Class `Point` had two *named instance variables*, `x` and `y`. The type and number of named instance variables is fixed when the class is defined. Objects may also contain *indexed instance variables*. These are accessed via a zero-based index. Each object may define a different number of indexed instance variables. The size of each variable is defined in the class header by the word `<Indexed`.

```
:Class Array <Super Object  CELL <Indexed

    At:  ( index -- value )  (At)  ;M
```

```
   To:  ( value index -- )  (To)  ;M

;Class
```

We have declared that an `Array` will have indexed instance variables that are each `CELL` bytes wide. To define an array, put the number of elements before the class name:

```
10 Array myArray
```

This will define an `Array` with ten elements, numbered from 0 to 9. We can access the array data with the `At:` and `To:` methods:

```
4 At: myArray .

64 2 To: myArray
```

Indexed instance variables allow the creation of arrays, lists, and other collections.

Early vs. Late Binding

In these examples, you may have been thinking, "All of this message sending must be taking a lot of time." In order to execute a method, an object must look up the message in its class, and then its superclass, until it is found.

But if the class of the object is known at compile time, the compiler does the lookup then and compiles the

Different Approaches to Object-Oriented Programming in Forth

There have been many different implementations of object-oriented programming presented over the years. I have found over half a dozen in back issues of *Forth Dimensions.* There are many different syntaxes, but in most cases the internals fall into one of three categories.

The first is pure *early binding.* The Forth vocabulary mechanism is used to resolve method names at compile time. Methods are just ordinary Forth words. This approach is often the simplest and has no run-time penalty. It is a solution to the name clutter and information-hiding problems of large programs, but the system is not as flexible as ones that use late binding.

The second approach is the *virtual method* technique. In this implementation, each object contains a pointer to a table of functions. A method references a specific offset into the table. When the method is used, it indexes into the method table and executes the code there. This technique has a small but constant overhead in method dispatch. It also has the advantage that all messages are late-bound and can be redefined by a subclass. The downside of this is that a method can only be used on objects that inherit from a common base class. If two different classes have a "print" method at a different offset, disaster awaits. This technique, by itself, does not solve the namespace problem, and so must be augmented with something usually based on search-order.

The third approach is a pure *message-sending* architecture. Either the objects or the messages are active, but the message binding is done at run time by some kind of lookup algorithm The advantage is that a message can be sent to any object that understands it, regardless of the inheritance tree. The primary disadvantage is the run-time overhead of message lookup. This can be overcome to some degree by using message caching and only doing a full search if the message is not in the cache. Also, if the message selector is a token instead of a name, we can avoid expensive string comparisons in the search.

The Yerk model is really a combination of the first and third styles. It will do early binding whenever possible, doing late-bound message sending only when necessary or desired. This gives the performance advantages of early binding with the flexibility of message sending. It also has a common feature missing in most of the object-oriented implementations I have seen: named instance variables that may themselves be objects. I think this is an essential feature for full use of the object-oriented programming style.

A Forth programmer can use object-oriented programming to supplement a normal Forth programming style, treating it as just another tool in the toolbox; or can buy into the technology whole-hog and use it for everything. An example of the latter is found in Mops, where objects and classes are used almost exclusively. Even basics such as the assembler and floating-point libraries are implemented in classes. Both approaches are useful, and sometimes an implementation custom-tailored for a specific application is required. This Forth is putty in our hands!

References

Object-Oriented Forth in Assembly
András Zsótér
FD Volume XVI, Number 6, March/April 1995

Object-oriented Forth
Rick Hoselton
FD Volume X, Number 2, July/August 1988

Yerk Comes to the PC
Rick Grehan
FD Volume XIII, Number 5, January/February 1992

Object-Oriented Forth
Roger Bicknell
FD Volume XIII, Number 5, January/February 1992

Simple Object-Oriented Forth
Clive Maynard
FD Volume XIII, Number 5, January/February 1992

execution token of the method. This is called *early binding*. There is still some overhead with calling a method, but it is quite small. In all the code we have seen so far, the compiler will do early binding.

There are cases when you do want the lookup to occur at run time. This is called *late binding*. An example of this is when you have a Forth variable that will contain a pointer to an object, but the class of the object is not known until run time. The syntax for this is:

```
VARIABLE objPtr   myPoint objPtr !

Print: [ objPtr @ ]
```

The expression within the brackets must produce an object address. The compiler recognizes the brackets and will do the message lookup at run time.

(Don't worry, I haven't redefined [or]. When a message selector recognizes the left bracket, it uses PARSE and EVALUATE to compile the intermediate code, then compiles a late-bound message send. This also works in the interpretive state.)

Class Binding

(Dave Boulton called this "promiscuous binding.")

Class binding is an optimization that allows us to get the performance of early binding when we have object pointers or objects that are passed on the stack. If we use a selector with a class name, the compiler will early bind the method, assuming that an object of that class is on the stack. So if we write a word to print a point like this,

```
   : .Point   ( aPoint -- )
       Print: Point ;

   objPtr @ .Point
```

it will early bind the call. If you pass anything other than a Point, you will not get the expected result (it will print the first two cells of the object, no matter what they are). This optimization technique should be used with care until a program is fully debugged.

Creating Objects on the Heap

If a system has dynamic memory allocation, the programmer may want to create objects on the heap at run time. This may be the case, for instance, if the programmer does not know how many objects will be created by the user of the application.

The syntax for creating an object on the heap is:

```
Heap> Point objPtr !
```

Heap> will return the address of the new point, which can be kept on the stack or stored in a variable. To release the point and free its memory, we use:

```
   objPtr @ Release
```

Before the memory is freed, the object will receive a Release: message. It can then do any cleanup necessary (like releasing other instance variables). This is similar to a C++ destructor.

Implementation

The address of the current object is stored in the value ^base. (In a native system, this would be a good use for a processor register.)

The only time you can use ^base is inside a method. Whenever a method is called, ^base is saved and loaded with the address of the object being sent the message. When the method exits, ^base is restored.

Class Structure

All offsets and sizes are in Forth cells.

Offset	*Size*	*Name*	*Description*
0	8	MFA	Method dictionary (eight-way hashed list)
8	1	IFA	Linked-list of instance variables
9	1	DFA	Data length of named instance variables
10	1	XFA	Width of indexed instance variables
11	1	SFA	Superclass pointer
12	1	TAG	Class tag field
13	1	USR	User-defined field

The first eight cells are an eight-way hashed list of methods. Three bits from the method selector are used to determine which list the method may be in. This cuts down search time for late-bound messages.

The IFA field is a linked list of named instance variables. The last two entries in this list are always self and super.

The DFA field contains the length of the named instance variables for an object.

The XFA field actually serves a dual role. For classes with indexed instance variables, it contains the width of each element. For non-indexed classes, this field is zero.

The TAG field contains a special value that helps the compiler determine if a structure really represents a class. In native implementations, a unique code field is used to identify classes, but this is not available in ANS Forth.

The USR field is not used by the compiler but is reserved for a programmer's use. In the future, I may extend this concept of *class variables* to allow adding to the class structure. This field is used in a Windows implementation to store a list of window messages the class will respond to.

Object Structure

Offset	*Size*	*Description*
0	1	Pointer to object's class
1	DFA	Named instance variable data
DFA+1	1	Number of indexed instance variables (if indexed)
DFA+2	?	Indexed instance variables (if indexed)

The first field of a global or heap-based object is a pointer to the object's class. This allows us to do late binding. Normally, the class field is not stored for an instance variable. This saves space and is not usually needed because the compiler knows the class of the instance variable and the instance variable is not visible outside of the class definition. For indexed classes, the class pointer is always stored because the class contains information needed to locate the indexed data.

When an object executes, it returns the address of the first named instance variable. This is what we refer to as the *object address*. This field contains the named instance variable data. Since instance variables are themselves objects, this structure can be nested indefinitely.

Objects with indexed instance variables have two more fields. The indexed header contains the number of indexed instance variables. The width of the indexed variables is stored in the class structure, which is why we must always store a class pointer for indexed objects.

Following the indexed header is the indexed data. The size of this area is the product of the indexed width and the number of elements. There are primitives defined to access this data area.

Instance Variable Structure

Offset	*Size*	*Name*	*Description*
0	1	link	points to link of next ivar in chain
1	1	name	hash value of name
2	1	class	pointer to class
3	1	offset	offset in object to start of ivar data
4	1	#elem	number of elements (indexed ivars only)

The link field points to the next instance variable in the class. The head of this list is the IFA field in the class. When a new class is created, all the class fields are copied from the superclass, so the new class starts with all the instance variables and methods from the superclass.

The name field is a hash value computed from the name of the instance variable. This could be stored as a string with a space and compile-time penalty, but with a good 32-bit hash function, collisions are not common. In any event, the compiler will abort if you use a name that collides with a previous name. You can rename your instance variable, or improve the hash function.

Following the name is a pointer to the class of the instance variable. The compiler will always early-bind messages sent to instance variables.

The offset field contains the offset of this instance variable within the object. When sending a message to an object, this offset is added to the current object address.

If the instance variable is indexed, the number of elements is stored next. This field is not used for non-indexed classes.

Unlike objects, instance variables are not names in the Forth dictionary. Correspondingly, you cannot execute them to get their address. You can only send messages to them. If you need an address, you can use the `Addr:` method defined in class `Object`.

Method Structure

Methods are stored in an eight-way linked-list from the MFA field. Each method is identified by a 32-bit selector which is the parameter field address of the message selector.

Offset	*Size*	*Description*
0	1	Link to next method
1	1	Selector
2	1	Method execution token

The code for a method is created with the Forth word `:NONAME`. In this implementation, it contains no special prolog or epilog code. When the method executes, the current object will be in `^base`. Method execution is done by the following word that saves the current object pointer and loads it from the stack, calls the method, and then restores the object pointer.

```
: EXECUTE-METHOD   ( ^obj xt -- )
    ^base >R
    SWAP TO ^base   EXECUTE
    R> TO ^base ;
```

When a method is compiled into a definition, the object and execution token are compiled as literals followed by EXECUTE-METHOD.

This represents the overhead for calling a method over a normal colon definition. (This was one of the concessions I made to ANS Forth. In the native versions, a fast code word at the start and end of a method performed a similar action, making the overhead negligible.)

When a message is sent to an instance variable, the method execution token and variable offset are compiled as literals followed by `EXECUTE-IVAR`:

```
: EXECUTE-IVAR   ( xt offset -- )
     ^base >R
     ^base + TO ^base  EXECUTE
     R> TO ^base ;
```

An optimization is made if the offset is zero (for messages to `self` and `super` and the first named instance variable). Since we do not need to change `^base`, we just compile the execution token directly.

Selectors are Special Words

In the Yerk implementation, the interpreter was changed (by vectoring `FIND`) so that it automatically recognized any word ending in : as a message to an object. It computed a hash value from the message name and used this as the selector. This kept the dictionary small.

In ANS Forth, there is no way to modify the interpreter (short of writing a new one). It has also been argued whether this is a good thing anyway.

In this implementation, message selectors are immediate Forth words. They are created automatically the first time they are used in a method definition. Since they are unique words, we use the parameter field of the word as the selector.

When the selector executes, it compiles or executes code to send a message to the object that follows. If used inside a class, it first looks to see if the word is one of the named instance variables. If not, it sees if it is a valid object. Lastly, it sees if it is a class name and does class binding.

Yerk also allowed sending messages to values and local variables, and automatically compiled late-bound calls. In ANS Forth, we cannot tell anything about these words from their execution token, so this feature is not implemented. We can achieve the same effect by using explicit late binding:

```
Message: [aValue]
```

Object Initialization

When an object is created, it must be initialized. The memory for the object is cleared to zero, and the class pointer and indexed header are set up. Then each of the named instance variables is initialized.

This is done with the recursive word `ITRAV`. It takes the address of an instance variable structure and an offset, and follows the chain, initializing each of the named instance variables in the class and sending it a `ClassInit:` message. As it goes, it recursively initializes that instance variable's instance variables, and so on.

Finally, the object is sent a `ClassInit:` message. This same process is followed when an object is created from the heap.

Example Classes

I have implemented some simple classes to serve as a basis for your own class library. These classes have names and methods similar to the predefined classes in Yerk and Mops. The code for the class implementation and sample classes is available from the Forth Interest Group's FTP site:

ftp://ftp.forth.org/pub/Forth/ANS/CLASS.ZIP

Conclusions

For me, the primary benefit of using objects is in managing complexity. Objects are little bundles of data that understand and act on messages sent by other parts of the program. By keeping the implementation details inside the object, it appears simpler to the rest of the program. Inheritance can help reduce the amount of code you have to write. If a mature class library is available, you can often find needed functionallity already there.

If the Forth community could agree on an object-oriented model, we could begin to assemble an object-oriented Forth library similar to the Forth Scientific Library project headed by Skip Carter—code and tools that all Forth programmers can share. That project had not been possible before the ANS standardization of floating point in Forth.

Unfortunately, there are many different ways to add objects to Forth. Just look at the number of articles on object-oriented programming that have appeared in *Forth Dimensions* over the past ten years. Because Forth is so easy (and fun) to modify and extend, everybody ends up doing it their own (different) way.

Andrew McKewan was introduced to Forth in college and was fascinated by it. He has been using Forth professionally and as a hobby for over ten years, and works for Finnigan Corporation where he writes instrument-control software for mass spectrometers. He can be reached by e-mail at his mckewan@austin.finnigan.com address.

CLASS.FTH — main implementation file

```
\ Object-oriented extensions from Yerk/Mops        Version 1.0, 4 Feb 1997
\ Andrew McKewan                                   mckewan@austin.finnigan.com

DECIMAL

\ ================================================================
\ Misc words. You may already have some of these.

: HAVE   ( -- f )  BL WORD FIND NIP ;

1 CELLS CONSTANT CELL
: C+!   ( char addr -- )   DUP C@  ROT +  SWAP C! ;

: RESERVE   ( n -- )  HERE OVER ERASE ALLOT ;

\ Build link to list head at addr
: LINK   ( addr -- )   HERE  OVER @ ,  SWAP ! ;

: NOOP ;

: BOUNDS   ( a n -- limit index )  OVER + SWAP ;

HAVE PARSE 0= [IF]
: PARSE   ( -- addr len )  BL WORD COUNT ;
[THEN]

\ ================================================================
\ Class Structure.

0 VALUE ^Class                  \ pointer to class being defined
0 VALUE newObject               \ object being created
0 VALUE (ClassInit:)            \ selector for ClassInit: message

\ The following offsets refer to the ^Class, or Pfa of the class.
\ The first 8 cells are the hashed method dictionary.
```

```
: IFA   8 CELLS + ;               \ ivar dict Latest field
: DFA   9 CELLS + ;               \ datalen of named ivars
: XFA  10 CELLS + ;               \ width of indexed area, <= 0 if not indexed
: SFA  11 CELLS + ;               \ superclass ptr field
: TAG    12 CELLS + ;             \ class tag field

13 CELLS CONSTANT classSize       \ size of class pfa

CREATE classTag                   \ contents of tag field for valid class

: ?isClass  ( pfa -- f )          \ is this a valid class?
      TAG @ classTag = ;

: ?isObj  ( pfa -- )              \ is this a valid object?
      @ DUP IF ?isClass THEN ;

: classAllot  ( n -- )            \ Allot space in the current class
      ^Class DFA +! ;

: classAlign  ( -- )              \ Align class data size (optional)
      ^Class DFA @  ALIGNED  ^Class DFA ! ;

: @width  ( ^class -- elWidth )   \ return the indexed element width for a class
      XFA @  0 MAX ;

\ Error if not compiling a new class definition
: ?Class   ^Class   0= ABORT" Not inside a class definition" ;

\ =====================================================================
\ Objects have a pointer to their class stored in the first cell of
\ their pfa. When they execute, they return the address of the cell
\ following the class pointer, which is location of the first named
\ instance variable.
\
\ Object structure: | ^class | named ivars |
\
\ If an object is indexed, an indexed header appears after the data area.
\ This header consists of a cell containing the number of elements.
\ The indexed data follows this header.
\
\ Indexed object:   | ^class | named ivars | #elems | indexed ivars |
\ =====================================================================

: (Obj)  ( -- )  CREATE  DOES> CELL+ ;

: >obj  ( xt -- ^obj )            \ get the object address from the execution token
      >BODY CELL+ ;

: >class  ( ^obj -- ^class )      \ get the class of an object
      CELL - @ ;

\ =====================================================================
\ Methods are stored in an 8-way linked-list from the MFA field.
\ Each method is identified by a 32-bit selector which is the parameter
\ field of the selector. Offsets are in cells.
\
\ Method Structure:
\      0   link to next method
\      1   selector
\      2   method execution token (called mcfa below)
\
\ =====================================================================
\ Find the top of the method link for a given selector.
\ The "2/ 2/" below is to get a better distribution if the selectors
\ are aligned values.
: MFA  ( SelID ^Class -- SelID MFA )
      OVER 2/ 2/ 7 AND CELLS + ;
```

```
\ Search through a linked-list of methods for the given selector.
: ((FINDM))  ( SelID MFA -- mcfa true | false )
      BEGIN @ DUP
      WHILE  2DUP CELL+ @ = IF  2 CELLS + ( mfca )  NIP TRUE  EXIT THEN
      REPEAT NIP ;

: (FINDM)   ( SelID ^Class -- xt )      \ find method in a class
      MFA ((FINDM)) IF  @ EXIT  THEN
      TRUE ABORT" Message not understood by class" ;

: FIND-METHOD   ( SelID ^obj -- ^obj xt )     \ find method in object's class
      TUCK >class (FINDM) ;

\ ===================================================================
\ Method execution. The current object address is store in the value ^base.
\ The object is only valid inside of a method definition. When we call a
\ method, we save the old object pointer and set it to the current object.
\ When the method returns, we restore the object pointer.

0 VALUE ^base                     \ pointer to current object

: EXECUTE-METHOD  ( ^obj xt -- ) \ execute method, saving object pointer
      ^base >R   SWAP TO ^base   EXECUTE   R> TO ^base ;

: EXECUTE-IVAR  ( xt offset -- ) \ execute ivar method at given offset
      ^base >R  ^base + TO ^base  EXECUTE  R> TO ^base ;

\ Wrap catch so that it preserves the current object
HAVE CATCH [IF]
: CATCH  ( -- n )  ^base >R   CATCH   R> TO ^base ;
[THEN]

\ ===================================================================
\ For late-bound method calls, we compile code to look up the method
\ at runtime and execute it. The syntax is:
\
\      Selector: [ object ]
\
\ The code between [ and ] must return an object reference. If the method
\ is not found in the class of the object, a runtime error occurs.
\ Because we use PARSE, [ and ] must be on the save source line.
\ ===================================================================

: (Defer)  ( ^obj SelId -- )  OVER >class (FINDM) EXECUTE-METHOD ;
: Defer,   ( SelId -- )  POSTPONE LITERAL  POSTPONE (Defer) ;

: Defered  ( SelId -- )          \ Compile or execute a defered message send
      >R  [CHAR] ] PARSE  EVALUATE  R>
      STATE @ IF  Defer,  ELSE  (Defer)  THEN ;

\ True if string is "[" to start defered message send
: ?isParen  ( str -- f )   1+ C@ [CHAR] [ = ;

\ ===================================================================
\ Hash function for instance variable names. The "32 OR" is for
\ case-insensitive names. The compiler will warn you if you have
\ a hash collision.

: HASH  ( addr len -- n )
      TUCK BOUNDS ?DO  5 LSHIFT  I C@ 32 OR XOR  LOOP
      DUP 0< IF EXIT THEN  INVERT ;

: hash>  ( -- n )  BL WORD COUNT HASH ;

\ ===================================================================
\ Instance variable consists of five cell-sized fields. The fifth field
\ is used for indexed ivars only. Offsets are in cells.
\
\     Offset   Name      Description
\     ------   ----      --------------------------------------
\        0     link      points to link of next ivar in chain
\        1     name      32-bit hash value of name
\        2     class     pointer to class
\        3     offset    offset in object to start of ivar data
\        4     #elem     number of elements (indexed ivars only)
\
\ In the stack diagrams, "ivar" refers to the starting address of this
\ structure.  The IFA field of a class points to the first ivar.
\ =================================================================

: iclass     ( ivar -- 'class )   2 CELLS +   ;
: @IvarOffs  ( ivar -- offset )   3 CELLS + @ ;
: @IvarElems ( ivar -- #elems )   4 CELLS + @ ;

\ =================================================================
\ Build SUPER and SELF pseudo ivars. These are always the last
\ two ivars in a class. When we define a class, we will patch the
\ class fields to the appropriate class and superclass.

CREATE ^Self
       0 ,                  \ link
       hash> self ,         \ name
       0 ,                  \ class
       0 ,                  \ offset

CREATE ^Super
       ^Self ,              \ link
       hash> super ,        \ name
       0 ,                  \ class
       0 ,                  \ offset


\ Create a dummy class that "object" inherits from.

CREATE Meta  classSize RESERVE

^Super   Meta IFA !         \ latest ivar
classTag Meta TAG !         \ class tag

\ =================================================================
\ Determine if next word is an instance var.
\ Return pointer to class field in ivar structure.

: vFind  ( str -- str false | ^iclass true )
       ^Class
       IF     DUP COUNT HASH ^Class IFA ((FINDM))
              DUP IF  ROT DROP  THEN
       ELSE   FALSE
       THEN ;

\ send ClassInit: message to ivar on stack
: InitIvar   ( ivar offset -- )
       OVER @IvarOffs + newObject +   ( ivar addr )
       (ClassInit:) ROT iclass @ (FINDM) EXECUTE-METHOD ;

: ClassInit  ( -- )  \ send ClassInit: to newObject
       newObject (ClassInit:)
       newObject >class (FINDM) EXECUTE-METHOD ;

\ =================================================================
\ ITRAV traverses the tree of nested ivar definitions in a class,
\ building necessary class pointers and indexed area headers.

: ITRAV   ( ivar offset -- )  >R  ( ivar -- )
       BEGIN  DUP ^Super <>
```

```
      WHILE  DUP iclass @ IFA @
             OVER @IvarOffs R@ + RECURSE  ( initialize ivar's ivars )

             DUP iclass @  ( get ivar class )
             DUP XFA @  ( needs class pointer? )
             IF    OVER @IvarOffs R@ + newObject +
                   ( ivar ^Class ivarAddr -- )
                   2DUP CELL - !                   \ store class pointer
                   OVER @width  ( indexed? )
                   IF     SWAP DFA @ +             \ addr of indexed area
                          OVER @IvarElems          \ #elems
                          SWAP !                   \ upper array limit
                   ELSE   2DROP
                   THEN
             ELSE  DROP
             THEN
             DUP R@ InitIvar                 \ send ClassInit:
             @                               \ next ivar in chain
      REPEAT R> 2DROP ;

\ ===================================================================
\ Compile an instance variable dictionary entry.

: <Var ( #elems ^Class | ^Class -- )
      BL WORD vFind ABORT" Duplicate instance variable"
      COUNT HASH                             \ get hash value of name
      ALIGN ^Class IFA LINK ,                \ link & name
      DUP ,                                  \ class
      DUP XFA @ IF  CELL classAllot  THEN    \ if indexed, allow for class ptr
      ^Class DFA @ ,                         \ offset to ivar data
      DUP @width DUP
      IF  ROT DUP ,  * CELL+  THEN           \ #elems, cell for idx-hdr
      SWAP DFA @ + classAllot ;              \ Account for named ivar lengths

\ ===================================================================
\ Build an instance of a class. If we are inside a class definition,
\ build an instance variable. Otherwise build a global object.

\ Compile the indexed data header into an object
: IDX-HDR   ( #elems ^Class | ^Class -- indlen )
      @width DUP IF  OVER , ( limit )  *  THEN ;

: (Build)   ( #elems ^Class | ^Class -- )
      ^Class
      IF     <Var                      \ build an ivar if we are inside a class
      ELSE
             (Obj)                     \ create object
             DUP >R ,                  \ store class pointer
             HERE TO newObject         \ remember current object
             R@ DFA @ RESERVE          \ allot space for ivars
             R@ IDX-HDR RESERVE        \ allot space for indexed data
             R> IFA @ 0 ITRAV          \ init instance variables
             ClassInit                 \ send CLASSINIT: message
      THEN ;

\ ===================================================================
\ Build a class header with its superclass pointer.

: :Class  ( -- )
      CREATE 0 TO ^Class
      DOES>  (Build) ;

: <Super  ( -- )
      HERE TO ^Class                   \ save current class
      classSize ALLOT                  \ reserve rest of class data
      ' >BODY                          \ pfa of superclass
      DUP ^Class classSize MOVE \ copy class data
      DUP ^Class SFA !                 \ store pointer to superclass
      ^Super iclass !                  \ store superclass in SUPER
      ^Class ^Self iclass ! ;          \ store my class in SELF

: ;Class  ( -- )
      classAlign                \ align class data size (optional)
      0 ^Super iclass !         \ clear out super and self class pointers
      0 ^Self  iclass !
      0 TO ^Class ;             \ clear class compiling flag

\ ===================================================================
\ Object Compiler. We rely on being able to classify the type of
\ object from it's execution token. There is no general way to
\ do this in ANS forth for builtin types such as VALUEs. So we
\ only allow message sends to objects and classes. In Yerk, the
\ following will send a late-bound message to a object pointer:
\
\      0 VALUE theObject  ' myObject TO theObject
\
\      Message: theObject
\
\ Here we will have to use the following syntax (which does the same
\ this and is also allowed in Yerk):
\
\      Message: [ theObject ]
\
\ Key to instantiation actions
\ 1 = objType      defined as an object
\ 2 = classType           as a class
\ 5 = parenType           open paren for defer group
\
\ ===================================================================

\ ( str -- xt tokenID )  Determine type of token referenced by string.
: refToken
      DUP ?isParen       IF  3 EXIT  THEN
      FIND 0= ABORT" undefined object"
      DUP >BODY ?isObj   IF  1 EXIT  THEN
      DUP >BODY ?isClass IF  2 EXIT  THEN
      TRUE ABORT" Invalid object type" ;

: (ivarRef)    ( xt offset -- )              \ compile ivar reference
      SWAP  POSTPONE LITERAL  POSTPONE LITERAL  POSTPONE EXECUTE-IVAR ;

: ivarRef    ( selID ^iclass -- )            \ compile ivar reference
      CELL+ FIND-METHOD  SWAP @  ( xt offset )  ?DUP
      IF     (ivarRef)
      ELSE   COMPILE,  ( optimize for offset zero )
      THEN ;

: callMethod   ( xt -- )                     \ compile method call
      POSTPONE LITERAL  POSTPONE EXECUTE-METHOD ;

: (objRef)    ( SelID objCfa -- )            \ compile object reference
      >obj FIND-METHOD SWAP ( xt ^obj )
      POSTPONE LITERAL  callMethod ;

\ ( selID $str -- )  Build a reference to an object or vector
: objRef
      refToken CASE
        1 ( object ) OF  (objRef)                          ENDOF
        2 ( class  ) OF  >BODY (FINDM) callMethod   ENDOF
        3 ( paren  ) OF  DROP Defered               ENDOF
      ENDCASE ;

\ ( selID $str -- )  Execute using token in stream
: runRef
```

```
     refToken CASE
       1 ( object ) OF  >obj FIND-METHOD                  ENDOF
       2 ( class  ) OF  >BODY (FINDM)            ENDOF
       3 ( paren  ) OF  DROP Defered EXIT        ENDOF
     ENDCASE  EXECUTE-METHOD ;

\ ================================================================
\ Selectors are immediate words that send a message to the object
\ that follows. The Yerk requirement that selectors end in ":" is
\ enforced here but not otherwise required by the implementation.

\ This is the message compiler invoked by using a selector.
: message   ( SelID -- )
     BL WORD  STATE @
     IF     vFind               \ instance variable?
            IF    ivarRef       \ ivar reference
            ELSE   objRef       \ compile object/vector reference
            THEN
     ELSE   runRef              \ run state - execute object/vector ref
     THEN ;

: ?isSel  ( str -- flag )       \ true if word at addr is a selector xxx:
     DUP DUP C@ CHARS +  C@ [CHAR] :  =  SWAP C@ 1 > AND ;

: ?Selector ( -- )              \ Verify that following word is valid selector
     >IN @  BL WORD ?isSel 0= ABORT" Not a selector"  >IN ! ;

\ Create a selector that sends a message when executed.
: Selector  ( n -- )  ?Selector
     CREATE IMMEDIATE  DOES> message ;

\ If the selector already exists, just return the existing selector,
\ otherwise create a new selector.
: getSelect  ( -- n )
     >IN @  BL WORD FIND
     IF     >BODY NIP  ( already defined )
     ELSE   DROP >IN ! Selector
            HERE
     THEN ;

Selector ClassInit:    getSelect ClassInit: TO (ClassInit:)

\ ================================================================
\ Build a methods dictionary entry. :M starts a method definition
\ by adding to the class method list and starting the compiler with
\ :NONAME. ;M ends a method and saves the method xt.

: :M   ( -- )
     ?Class
     getSelect
     DUP ^Class MFA ((FINDM))              \ is method already defined?
     IF
\           CR ." Method redefined "
\           HERE COUNT TYPE SPACE
\           ^Class BODY> >NAME .ID         \ print class name

            ^Class U> ABORT" Method redefined in same class"
     THEN
     ALIGN               \ align method
     ^Class MFA LINK     \ link into method chain
     ( SelID ) ,         \ name is selector's hashed value
     HERE 0 ,            \ save location for method xt
     :NONAME  ;          \ compile nameless definition

: ;M   ( -- )  \ end a method definition
     ?Class  POSTPONE ;  SWAP ! ( save xt )  ; IMMEDIATE
```

```
\ ================================================================
\ Build a new object on the heap for class. Use: Heap> className
\ gets heap, and returns ptr. Throws an error if not enough memory

HAVE ALLOCATE [IF]

: ?MEMERR  ( ior -- )  ABORT" Memory allocation error" ;

Selector Release:   \ sent to an object before it is freed

: allocObj  ( size class -- )    \ allocate object and store in newObject
      OVER CELL+                 \ allow for class ptr
      ALLOCATE ?MEMERR           \ ( size class addr -- )
      DUP CELL+ TO newObject     \ object address
      !                          \ create the class ptr
      newObject SWAP ERASE ;     \ clear to zero

: (heapObj)  ( #elems class | class -- ^obj )
      >R  ( save class on return stack )
      R@ DFA @  ( ivar data size )
      R@ @width ?DUP
      IF            \ indexed object, add size of indexed area
                    \ ( #elems size width -- )
             2 PICK * + ( indexed data )  CELL+ ( idxHdr )
             R@ allocObj
             newObject R@ DFA @ + !  ( store #elems in idxHdr )
      ELSE
             R@ allocObj         \ non-indexed object
      THEN
      R> IFA @ 0 ITRAV           \ initialize instance variables
      ClassInit                  \ send ClassInit: message to new object
      newObject  ;               \ return object address

: HEAP>   ( -- addr )
      ' >BODY  DUP ?isClass 0= ABORT" Not a class"
      STATE @
      IF    POSTPONE LITERAL  POSTPONE (heapObj)
      ELSE   (heapObj)
      THEN ; IMMEDIATE

: RELEASE  ( ^obj -- )           \ free heap object
      Release: [ DUP ]           \ send Release: message to object
      CELL - FREE ?MEMERR ;      \ deallocate it

[THEN]

\ ================================================================
\ Support for indexed instance variables. When object of these classes
\ are defined, the number of elements should be on the stack.

\ Set a class and its subclasses to indexed

: <Indexed  ( width -- )  ?Class  ^Class XFA !  ;

\ For some classes, we always want the class pointer to be stored whenever
\ the object is an instance variable, so that the object can receive late-
\ bound messages. This is already the case for indexed classes. Here we
\ fake it out by storing -1 into the indexed field. This is the reason
\ for the "0 MAX" in the definition of @width.

: <General  ( -- )  -1 <Indexed ;

\ self returns the same address as ^base, but is used for late bound calls
\ to the object, i.e.   Selector: [ self ]

: self  ( -- )      ?Class
      ^Class XFA @ 0= ABORT" Class must be <General or <Indexed"
      POSTPONE ^base ; IMMEDIATE
```

```
\ =================================================================
\ Indexed primatives. These should be in code for best performance.

: idxBase   ( -- addr )           \ get base of idx data area
     ^base  DUP >class DFA @ +  CELL+ ;

: limit     ( -- n )              \ get idx limit (#elems)
     ^base  DUP >class DFA @ +  @ ;

: width     ( -- n )              \ width of an idx element
     ^base >class XFA @ ;

: ^elem   ( index -- addr )       \ get addr of idx element
     width *  idxBase + ;

\ Fast access to byte and cell arrays.
: At1  ( index -- char )   idxBase + C@ ;
: At4  ( index -- cell )   CELLS idxBase + @ ;

: To1  ( char index -- )   idxBase + C! ;
: To4  ( cell index -- )   CELLS idxBase + ! ;

: ++1  ( char index -- )   idxBase + C+! ;
: ++4  ( cell index -- )   CELLS idxBase + +! ;

\ Compute total length of object.
\ The length does not include class pointer.
: objlen  ( -- objlen )
     ^base >class DUP DFA @  ( non-indexed data )
     SWAP @width ?DUP
     IF  idxBase CELL - @ ( #elems ) * +  CELL+  THEN ;

\ =================================================================
\ Runtime indexed range checking. Use +range and -range to turn range
\ checking on and off.

: ?range  ( index -- index )            \ range check
     DUP idxBase CELL - @ ( #elems )  U< IF EXIT THEN
     TRUE ABORT" Index out of range" ;

0 VALUE (?idx)                          \ execution vector for range checking

: ?idx   (?idx) EXECUTE ;

: +range  ['] ?range TO (?idx) ;  +range
: -range  ['] NOOP   TO (?idx) ;

\ =================================================================
\ Primatives for cell-sized objects.

: M@  ( -- n )  POSTPONE ^base  POSTPONE @ ; IMMEDIATE
: M!  ( n -- )  POSTPONE ^base  POSTPONE ! ; IMMEDIATE

\ =================================================================
\ Define base class "object" from which all other classes inherit
\ Some of the common indexed methods are defined here.

:Class Object  <Super Meta

      :M Class:   ^base >class  ;M          \ non-IX - leave class ptr
      :M Addr:    ^base         ;M          \ get object address

      \ ( -- len )  Return total length of object
      :M  Length:    objlen      ;M

HAVE DUMP [IF]
      :M  Dump:     ^base objlen DUMP  ;M
[THEN]
```

```
      :M  Print:     ." Object@" ^base U.  ;M

      :M ClassInit:   ;M        \ null method for object init
      :M Release:     ;M        \ null method for object release

;Class

\ Bytes is used as the allocation primitive for basic classes
\ It creates an object of class Object that is n bytes long.
\ You can get the address by sending it an addr: message.

: bytes   ( n -- )
      ?Class  ['] Object >BODY <Var  classAllot ;


\ ================================================================
\ Load primative classes and test routines.

S" VAR.FTH"    INCLUDED
S" ARRAY.FTH"  INCLUDED
S" TEST.FTH"   INCLUDED

CR .( Classes loaded )
```

VAR.FTH — cell-sized classes

```
\ Basic object variables
\ Version 1.0, 4 Feb 1997
\ Andrew McKewan
\ mckewan@austin.finnigan.com

\ ================================================================
\ Define the basic cell-sized variable class. This is a generic superclass
\ that defines the basic access operators.

:Class CellObj  <Super Object

      CELL bytes Data

      :M  Get:   ( -- n )   M@   ;M
      :M  Put:   ( n -- )   M!   ;M

      :M  Clear:   0 M!  ;M
      :M  Print:   M@ .  ;M

      \ ( ^obj -- )  copies data from another CellObj
      :M ->:   @ M!  ;M

;Class

\ ================================================================
\ Var is for integer data

:Class Var   <Super CellObj

      :M  +:  ( n -- )        ^base +!  ;M
      :M  -: ( n -- )  NEGATE ^base +!  ;M

      :M  *:  ( n -- )  M@  *       M!   ;M
      :M  /:  ( n -- )  M@  SWAP /  M!   ;M
      :M  Negate:    M@  NEGATE  M!   ;M
```

```
;Class

\ ======================================================================
\ Bool is for storing booleans. It always returns TRUE or FALSE.

:Class Bool   <Super CellObj

       \ Put: always stored Forth boolean flag
       :M Put:  ( f -- )   0= 0= M!  ;M

       :M Set:     TRUE  M!  ;M
       :M Invert:   M@ 0= M!  ;M

       :M Print:  M@ IF ." true " ELSE ." false " THEN  ;M
;Class

\ ======================================================================
\ ExecVec stores an execution token.

:Class ExecVec   <Super CellObj

       \ Execute xt stored in variable
       :M Exec:  ( -- )   M@ EXECUTE  ;M

       \ Initialize to do nothing
       :M Clear:      ['] NOOP M!  ;M
       :M ClassInit:  Clear: self  ;M

;Class

\ ======================================================================
\ Ptr stores a pointer to dynamically-allocated memory. We also keep track
\ of the current size of the memory block.

HAVE ALLOCATE [IF]

:Class Ptr   <Super CellObj

       Var size                  \ current size

       :M Size:  ( -- n )        \ get current size
              Get: size  ;M

       :M Release:  ( -- )       \ release current memory
              M@ IF  M@ FREE ?MEMERR  0 M!  THEN   Clear: size  ;M

       :M New:  ( len -- )       \ create a new memory block
              Release: self  DUP ALLOCATE ?MEMERR  M!  Put: size  ;M

       :M Resize:  ( len -- )    \ resize memory block
              M@ OVER RESIZE ?MEMERR  M!  Put: size  ;M

       :M Nil?:  ( -- f )        \ true if no memory has been allocated
              M@ 0=  ;M

;Class

[THEN]
```

(Code continues in the next issue.)

Yet Another Modest Proposal

Richard Astle
Del Dios, California

There was a lot of talk recently on comp.lang.forth and at FORML about object-oriented programming and object-oriented extensions for Forth. The consensus was, as usual in this community, that there is no consensus, which is perhaps as it should be: we're nothing if not experimentalists, and we're still in the process of that. As is the rest of the world for that matter, as shown by the disparate object models of Delphi and Java (both straightforwardly object-oriented languages, unlike the mixed-breed C++) among currently popular languages. For myself, in Forth, I'm quite happy with the Smalltalk/Neon/Yerk-inspired object model in Win32Forth, which may well become a de facto standard, or at least a model to compare other models with.

However, I would like to take a step back, to a time before inheritance and polymorphism, and reconsider data hiding, the oldest of the three tines of the object-oriented fork. I'm certain what I propose here is not an original idea in the Forth community, but it's not talked about much, and hasn't been standardized, so I think it's worth bringing up.

The consensus was that there is no consensus, which is perhaps as it should be.

For years, I've been using an idea inherited from Modula-2, Ada, and similar languages, involving "modules" of source code containing "implementation" and "interface" parts, "private" and "public" definitions. This structure, which will be familiar to many of you, is orthogonal to the "vocabulary" concept Forth has almost always had, in that, while VOCABULARY (now WORDSET) allows partitioning of definitions into separate name spaces, it does not promote modularization in the sense of grouping related words together in the source stream. This scattering of source is useful in some places (for an ASSEMBLER vocabulary, for example), but it's not very good packaging, and (since my assembler is only available in the metacompiler anyway) I've stopped using the vocabulary/wordset idea entirely.

My MODULE words come in two flavors, and I use both of them, depending on the situation. First is the set PRIVATE ... PUBLIC ... END-MODULE, used in that order in the stream of source code. PRIVATE introduces a section of source which contains definitions that will be invisible to words defined after the corresponding END-MODULE. Meanwhile, in the portion of the source code stream between PUBLIC and END-MODULE, the PRIVATE definitions are available, just as though PRIVATE never happened.

The way this is done, of course, is by manipulating headers: code, list, and data (if you have all those parts and distinctions) compile as usual. My implementation relies on manipulating a traditional, linear Forth dictionary structure (I use a Forth with separated headers and no hashing), but I'm assured by those who favor it that a hashing implementation is not difficult to conceive. In my environment, PRIVATE and PUBLIC are implemented by manipulating the header dictionary pointer, and END-MODULE patches a header link. Specifically, PRIVATE saves the value of HDP and moves it ahead, leaving a gap of 1K or so (this gap often has to be adjusted, which is an annoyance and an ugliness in the implementation). Then names are added as usual until PUBLIC moves HDP back to where it was just before PRIVATE, without unlinking the private names. Between PUBLIC and END-MODULE, names are still added as usual, filling the gap, until END-MODULE sets the link in the first PUBLIC word to point to the last header before PRIVATE, effectively sealing off the headers defined between PRIVATE and PUBLIC. Since the place where these headers exist is beyond the current HDP, their space is reclaimed in the normal course of adding more headers to the dictionary.

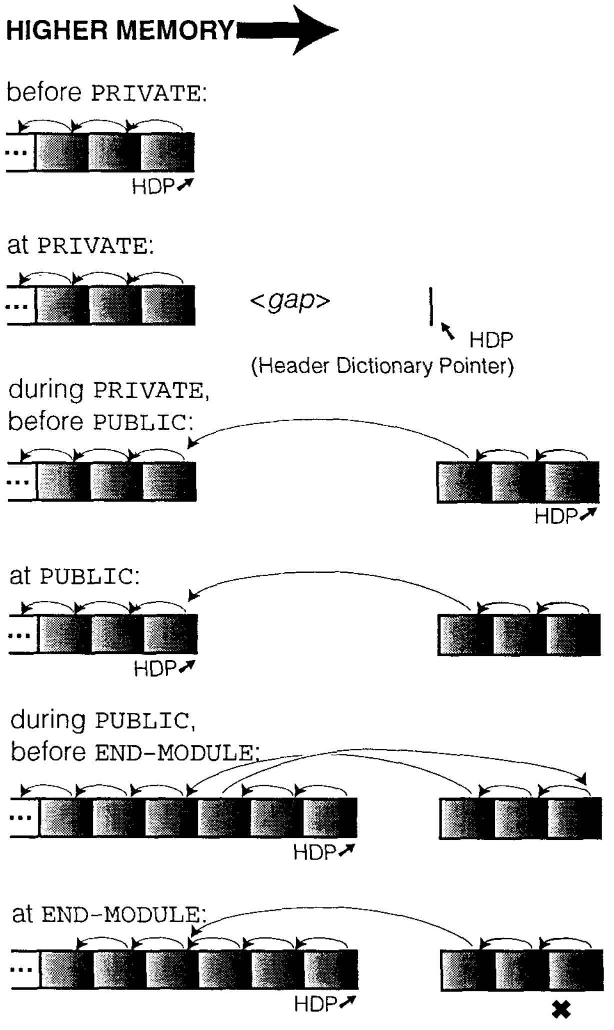
With this implementation, modules are nestable: a module can be placed in the PRIVATE section of another module, so long as there is still header space left to allow the header DP to be moved ahead. I have a large application, and though my headers share a 64K segment only with stacks, I have to be careful, fine-tuning the memory gap PRIVATE implements on a module-by-module basis. In a 32-bit Forth, things would not have to be quite so ugly.

My other version of module-defining words is the set MODULE ... EXPORTS ... END-MODULE. MODULE is just a synonym for PRIVATE—the difference is the word EXPORTS. In this version, all definitions in the module come in the source stream between MODULE (a.k.a. PRIVATE) and EXPORTS, while between EXPORTS and END-MODULE appears a list of words to be promoted to public status. My Forth has a word ALIAS, which creates a new header for an old word. Since I have separated headers, all this amounts to is taking the code pointer (it would be the list pointer in an indirect-threaded implementation) from the old header and putting it in the appropriate place in the new header, which means there is no execution-time speed penalty. EXPORTS executes the functional part of ALIAS over and over until it reaches END-MODULE. This method allows public words to be used in the definition of private words in the module, a feat which would otherwise require deferred words in a traditional, single-pass, Forth compilation model.

I came to this technique because I could, of course, but also because I needed it: I had too many headers, and I didn't like the technique of storing some of them on disk (which I inherited from Pierre Moreton) or of limiting names to three or five or eight significant characters. But it's not only header dictionary space that this technique conserves, but also my own conceptual space: I can reuse meaningful private names and not worry about name collisions. (My interpreter tells me when I'm redefining something, and I take steps to eliminate the clash. Redefinition is a time-honored Forth technique, but it can cause problems I'd rather do without.) When I list the dictionary, it's not cluttered up with quite so many implementation words, given names because they were once convenient factors.

A second advantage of this construction is that it promotes modularization: related words get clustered together. I don't find myself defining something in screen 5 just so I can use it in screen 500—I define it in screen 499, where it belongs. Theory aside, this makes things more manageable: if a word is defined private to a module and you want to change it, you don't have to do a full-source search to be sure you won't break something in a distant part of your code. As I noted earlier, this goes counter to the way VOCABULARY works, which allows grouping together words defined in distant reaches of source. I wouldn't propose getting rid of the VOCABULARY or WORDSET idea, but I don't find much use it anymore.

Both of these features are also advantages of fully object-oriented systems, which also have inheritance, polymorphism, and encapsulation (creating new data types along with the operations on them). Those are not esoteric issues anymore, but for us they're still under construction. For me, the usefulness of source code modules as I've described them here is obvious. I think any reasonably experienced Forth programmer can implement this construct. If I'm right, many will, or already have, and it will become a standard tool—whether it's ever Standard or not.



✖Even though these headers are still linked together and point back into the main dictionary, they're abandoned by the system, since nothing points *to* them—and HDP will, in due course, march right through their territory.

I came to this technique because I could, of course, but also because I needed it...

Richard Astle used Forth for a living for thirteen years, most of that time developing and maintaining a large planning tool and database-management system for the food service industry. In the process, he became familiar with many aspects of Forth language and development, some of which have been reported in these pages. He is now off to adventures with Delphi and Java, hoping not to forget his roots. He has a bachelors degree in mathematics from Stanford University, a master's in creative writing from San Francisco State, and a Ph.D. in English literature from the University of California (San Diego).

A Case in Point

MPE's Forth Coding Style Standard

[Portions of this document, including parts of some of the examples, were edited lightly by FD *for publication in this format. —Ed.]*

Introduction

This Forth layout standard covers the layout of Forth source code in text files. It covers the layout of code and comments, along with the use of the file, and the reasons for a standard, and the reasons for certain decisions and recommendations in the standard. This document reflects current programming practice at MPE, and was heavily influenced by the *MartelForth* layout specification produced by Laurie Newell of British Telecom, which is copyrighted by British Telecommunications, 1987.

This standard presents an approach to thorough coding and layout. One of its main threads is consistency. The key features for a standard are:

- Consistency from one programmer
- Consistency between many programmers
- Easy to follow
- Easy to understand
- Code which is easy to read and understand
- Code which is difficult to get wrong because of layout
- A layout which is also pleasant
- Unambiguous

Why a Standard?

There are several reasons for producing and following a standard for anything. This standard is produced for the following reasons.

In any organisation where many programmers use the same language, they will inevitably share source code, or work in teams on a given project. If every programmer writes code to their own standard, then communication will be difficult, and programmers will tend to prefer to 're-invent the wheel' rather than try to understand and use another's code. This is costly, time-consuming, and unreliable. By following a standard—any standard—code will be meaningful and maintainable by everyone.

When a programmer starts to learn a language, he or she will not know how best to either write or lay out the code being written. Books abound to teach the syntax, structure and word-sets of Forth, but there is little advice given on layout and practice. This document seeks to offer the collected experience of staff at MPE so that other programmers learn an accepted and clear standard.

However, the standard as presented here should not be taken as an absolute diktat. If an organisation has its own specific layout or documentation requirements, these should not be ignored. A standard is there to help, not hinder in the production of Forth code.

MicroProcessor Engineering Ltd.
133 Hill Lane - Shirley
Southampton SO15 5AF
England

Implications of Editors, Monitors, Printers

The use of an editor has many implications. One of these is the amount of code visible on the monitor at any one time. Another is the use of tabs and spaces in white space. Yet another is the decision of how the code is fragmented—how many words on a page or in a file—and how many files in the application or project.

The number of lines visible on the screen, coupled with the speed of cursor movement almost dictates the size of any word or procedure produced. This is of direct relevance to the style of code eventually written: whether the code is very vertical with lots of white space, or is very horizontal with much code on every line. This is discussed later.

The use of tabs to space out code and comments has a direct relevance to the amount of white space between elements of the source file. If spaces must be used, there will be little white space as its production is tedious. If tabs are used, there may be any amount of white space in the file, and it will be uniformly laid out, but the layout will be at the whim of any tool used with the file. A good example of this is the fact that DOS's tabs are preset to 8 columns (1,9,17, etc.) for programs such as PRINT and TYPE, but editors generally have programmable tab-stops.

It is therefore important to consider the tab spacing used in conjunction with the tools likely to be used with the source files. If an editor is capable of smart-indenting, the amount of indent has to be considered. If the indent is set too deep, very few structures will be easily nestable, but if the indent is too small, then the indent will not stand out as such.

The decision of how many words will be placed on one page or in one file depends on the nature of the compiler and editor in use. If the editor can have many windows onto many files, then many files may be used easily—with the code factored out on a per-file basis. However, if the editor does not support multiple files, then there will be a tendency to place all code in one file. If the compiler or editor does not support page-breaks (see later), the source code will tend to be fairly monolithic, with words simply in sequence. However, if the tools support paging then the code will tend to be grouped in pages, and structured accordingly.

MPE house rule:

Tabs will be set to eight characters (1,9,17 ...). Editors will be set to produce hard tabs. If you cannot discipline a programmer to set tabs to eight characters, then the editor must be set to convert tabs to spaces.

Horizontal/Vertical Layout: a Discussion

There are two main styles of source-code layout in use. One is vertical, as used in assembler source, and the other is horizontal, as used by most C programmers (and others).

Vertical code:

```
mov ax, bx
add bx, 3
inc di
```

Horizontal code:

```
for (i=0; i++, i<<=20) printf ("%d", I);
```

The horizontal layout leads to a high code density and minimal eye movement to read. The vertical layout encourages in-line comments and improved visibility due to white space. Both have their benefits and disadvantages.

Forth programmers usually prefer in-line comments alongside the definition of a word. This leads to more comprehensible code as it is both read and written, but relies on a rather vertical code layout. However, the novice programmer is likely to extend the vertical layout to the extreme of a typical assembler layout, and thus lose the high-level structure and flow of Forth source.

This generally derives from the phrasing of code—writing meaningful phrases or fragments of code on one line such that the comment describes the overall effect of the line, not the actions of individual words such as @ and +. Phrasing also helps point out code fragments which could be written more efficiently by being factored as separate words.

Making Forth code legible is a compromise between vertical and horizontal layouts—with code well phrased or factored, but with structures spread out for easy checking or modifying. As discussed in the section 'Control Structure Layout', however, even these are open to debate, as short bodies within structures are often best on the same line as the entire structure.

Comments: a Discussion

There have been two standards of Forth source code comments. One is the in-line comment, the other is an additional block of comment before or after in the file, or in another file.

The former has the advantage of being parallel to the code to which it relates whilst the latter reads as consistent English, or other human language, and is more descriptive. In a text file, both forms of comment may be supported. Because the page is at least 80 characters wide, each line may include a good in-line comment. Because the monitor screen is at least 25 lines deep, a definition may also have a 'header block' of comment above or below it. This comment could potentially be in another file, perhaps a documentation file, but is far more relevant and useable if it is in the same file, and on the same page as the code it explains.

Wisdom should be used in the wording in comments. The comment should not be so trivial that it is pointless ('fetch the contents of the variable'), but should not be so removed from the code that it conveys no information ('reads data structure'). It should indicate clearly, assisting the Forth itself ('get the pointer value'):

```
: EXAMPLE       \ – ; comments example
                \ pointless ...

  DATA 4 + @    \ get contents of variable
                \   no information ...
                \ read data structure
                \   useful ...
                \ get the pointer value
  @ EXECUTE
;
```

Comments should provide effective and efficient communication. Be aware of the amount and intention of your comments.Do not use comments to explain tricky code - rewrite it instead. Comments are not an alternative to the code, they should illuminate reason and intention. They are used by the reader to provide focus and understanding.

It is not necessary to comment every line explicitly, but all code should be commented.

The Layout of This Document

Within this standard, a description of the requirement of the standard will be followed by an example. In the example, the relevant code fragments will be printed in UPPER CASE and a fixed width font. The comments in the example will be in lower case. However, the case of the characters used in real code is not specified. Some compilers may be case-sensitive, others not. Also, different programmers may have case preferences. The code examples in this document will assume that the left-hand end of the line comes directly below the left-hand end of this type of text. Indented code will be relative to this column position:

```
\ the beginning of the line
  \ indented by one indent
```

File layout

Definition of a Page

A page is defined as the characters between Form Feed characters (ASCII 12d)—called 'Form Feed', but well known as 'page break'. These are the characters inserted into a file by the 'New page' functions of editors such as Word Perfect's *Program Editor*, MPE's *TED* and *WinTED*, or the Forth macros for *BRIEF*. The text from the start of the file up to the first page break is called the first page, not the zeroth page.

First page

The first page of a file should include any copyright, author, or other specific information. This may also include project details or other information relevant to the use of the code. If there are specific hardware dependencies, these should be outlined on the first page—this page is the one usually first seen when the file is browsed or edited.

If the compiler is capable of loading specific pages, or of skipping the remainder of a page, then it may be made to do so, else, each line of text on the first page will have to be commented out (with a \ comment word) [see Figure One].

Notice that the first line of the second example still contains a \ comment. This allows the first page to have a first-line comment much like all the other pages in the file. Because a tool often exists to display all the first-line comments, it is worth the first page also having one. See the section on "Page Layout."

Change History

It is often very useful to be able to track all changes to a file. This will be particularly important where a version control system is not being used, or where changes have to be made off-site, for example when commissioning equipment.

Change history at MPE is included on the first page in the following form

```
SFP001   date   description

PNB002   date   description
```

The first column contains the authors initials followed by a sequential, three-digit number. The second contains the date in the form 29 Jan 95 because this format causes the least confusion in international projects, and the third column contains a description of the changes. *All* dates should be in this form. The code is then commented with these markers

```
   changes          \ added PNB002
\ SFP001 ... changes start
   ...
\ ... SFP001 changes end
```

The use of a special comment CH\ may be useful to allow the change history markers to be stripped out periodically when a major release is issued. This prevents the cource code becoming very untidy.

Version Numbering

MPE has adopted the following version-numbering scheme that allows for the requirements of the office administration and for the use of automated tools such as build numbering schemes that increment automatically whenever the code is recompiled.

```
Version = X.YY.ZZZ
```

- *X* is the *Major Version* which changes for a major revision of the requirements or design specifications. For tools, the Major Version should always be incremented when previous code may be broken.
- *YY* is the *Minor Version* which changes when a feature or function is added or removed.
- *ZZZ* is the Release or Build number which changes for each maintenance release, for example for bug fixes and documentation improvements.

Figure One. In-house template for first page.

```
\ Project: Code for Front panel control
\ Customer: xxxxxxxx
\ Project: yyyyyyyy
\ Dependencies and requirements


\ (c) MicroProcessor Engineering Ltd and xxxxxxxxx
\ 133 Hill Lane<R>\ Southampton
\ SO15 5AF

\ Phone (01703) 631441


\ Note: this code requires the zzzzzzzz
\ interface card, and MPE 8031 Cross-compiler
\ and interrupt handling code.
\ Revision history goes here

or

\ Project: Code for Front panel control
PTO                       \ skip to next page
Customer: xxxxxxxx
Project: yyyyyyyy
Dependencies and requirements

(c) MicroProcessor Engineering Ltd and xxxxxxxxxx
133 Hill Lane
Southampton
SO15 5AF

Phone (01703) 631441

Note: this code requires the zzzzzzzz
interface card, and MPE 8031 Cross-compiler
and interrupt handling code.

Revision history goes here
```

The version number of the software should appear on all copies of master and issue disks and in any accompanying documentation.

A software package may consist of many components, which can be separately identified and will retain their own numbering. However, the package as a whole will have its own version number, and documentation identifying the components. We use a file called RELEASE.TXT or RELEASE.DOC for each component (held in separate directories), and PACKAGE.TXT or PACKAGE.DOC for the overal package.

Dependencies

A package should identify what is required to build or run this package, and what versions are required. These are all the hardware, software, and documentation items that the package depends on to run.

The Rest

The rest of the file, either the code after the project information, or the subsequent pages, will contain the code for the project, or the portion of the project.

It is normal to split the code for an entire system into several files. These are normally grouped into related areas: all the data structures in one file, all the serial i/o in another, and so on. If the project is small enough not to warrant many source files, then the code areas will be grouped on different pages: page 2 for data, page 3 for I/O, etc. This modularisation may be thought of as an impact of design on coding.

Each page should follow the layout described in detail in the following section.

If the compiler does not 'understand' pages, then pages may be simulated within the file. A method of so doing might be to have a pair of lines:

```
\ -- End of Page --
\
```

which are inserted at the end of every "page." It is then possible to use the editor's search function to find the next page. We much prefer the use of hard pages rather than having to simulate page breaks.

Page Layout

The Contents of a Page

All words defined on a single page should be related in function or area of the application. This allows the programmer to look easily at all related functions, without much need to cursor or search through the file.

First Line

The first line of the page should contain the name(s) of the words defined on the page, and the date/name stamp of the author. These will be in the form of a whole-line comment:

```
\ - R.G. - 30/10/91 - word1 word2 word3
```

This top-line identifies the author, the date the page was created or edited, and the names of the words—`word1`, `word2`, and `word3`—on the page. A block comment with narrative is also very useful for each group of words.

It will be noted that the stamp is at the beginning of the line. This simplifies the design of editor functions or macros which automatically insert the information on the top line of the page. It is recommended that such an editor be used—it will save typing.

Base and Numbers

If the number base for the code on a page is important, the base should be specified at the top of the page.

```
\ - R.G - 30/10/91 - word1

HEX

: WORD1
;
```

If the base changes again on the page, the new base should be specified in much the same way:

```
: WORD1
;

DECIMAL

: WORD2
;
```

A blank line is left either side of the base specification for reasons of visibility. The code may all be squeezed together, but part of the reason for having a standard is that the code might be legible and easy to understand.

Many compilers also allow the base to be specified as the number is typed:

```
#100        \ decimal 100
$100        \ hex 100 = decimal 256
%100        \ binary 100 = hex/decimal 4
```

If the compiler to be used supports this feature, then it is good practice to use it, as there can then be no mistake which number is meant at any time. If the compiler does not support the temporary base definition, then it is best to always prefix a hex number with a zero:

```
HEX<R>0100      \ hex 100 = decimal  256
0ADD            \ hex ADD = decimal 2781
ADD             \ the word 'ADD'
```

Debugging faults caused by search order and base failures can be lengthy and frustrating.

The tendency at MPE is to move away from regular base changes to the use of the prefix system described above.

MPE house rule

Where the base can be confused, you must use a base indicator at the start of the code section. Hexadecimal numbers *must* be preceded by a leading zero. Use of numeric literals is deprecated at MPE. Use constants or equates instead. The code will be *much* easier to maintain.

(Continued in the next issue.)

Using Forth to manipulate the real world

Forthware

A Gentle Introduction to Digital Filters

Skip Carter
Monterey, California

Introduction

With this month's column we will take an introductory look into a topic that many find to be intimidating: digital filters. Digital filters provide a means to condition a digital signal in order to achieve a variety of purposes. Depending upon the problem one is confronted with, a filter may be needed to solve one or more signal conditioning requirements: reduce unwanted noise, isolate or reject a piece of the signal, enhance certain components of a signal (e.g., its amplitude or temporal resolution). What makes digital filtering hard to master is that to create a filter to achieve your goals takes a bit of calculus (usually in the complex plane) to do it properly. What makes digital filtering intimidating is the degree to which various authors handle the calculus. If they hide the calculus, digital filters take on the air of a black art. If they just do the calculus, they throw their readers into the deep end and risk losing a few in the process. I will *try* to strike a balance and give you the math—but starting at the shallow end of the pool.

I learned digital filtering from exploration seismologists (the people who inject signals into the ground with various kinds of mechanical vibrators, thumpers, and explosives, in search of mineral and oil deposits). Other than notation, the mathematics behind the seismologists' version of digital filters is the same as the electrical engineers' version. What *is* different is that the two groups approach the subject somewhat differently. We will use the seismology approach; to the electrical engineers reading this, it will look odd at first (but *not* unfamiliar) but, ultimately, we get to the same end.

The Z Transform

We will start by defining a transform function that, at first, looks so trivial that it appears it cannot possibly be useful. Let us assume that data is digitized at a uniform sampling rate. Say our data consisted of the sequence of points:

$x_t = (10, 5, 0, 4, -3, -6, 0, \ldots)$ (1)

We can represent this data as a *polynomial:*

$X(Z) = 10 + 5Z + 0Z^2 + 4Z^3 - 3Z^4 - 6Z^5 + 0Z^6 + \ldots$ (2)

This polynomial is called the *Z transform* of the time series. The trick with Z transforms is how we interpret what the *Z* actually represents. The *Z* is called the *unit delay* operator because of the following: if we take (2) and multiply it by *Z* we get,

$Y(Z) = ZX(Z) = 10Z + 5Z^2 + 0Z^3 + 4Z^4 - 3Z^5 - 6Z^6 + 0Z^7 + \ldots$ (3)

which becomes the untransformed series:

$y_t = (0, 10, 5, 0, 4, -3, -6, 0, \ldots)$

which is just the original time series delayed by one unit of time. (Be careful, if you try to look the *Z* transform up in a book. In a variation of the pi-throwing contest I mentioned a few columns back, different authors use different sign conventions on the Z transform polynomial.)

This innocent-looking transform and delay operator turn out to be very powerful. Consider the following: suppose x_t represents the response of the ground to an explosion (which we will call the *impulse response).* If another explosion occurred ten time units later, the signal we would see is,

$y_t = (10, 5, 0, 4, -3, -6, 0, 0, 0, 0, 10, 5, 0, 4, -3, -6, 0, \ldots)$

which can be represented by,

$Y(Z) = X(Z) + Z^{10}X(Z)$

If the second explosion was half the strength of the first, we'd write,

$$Y(Z) = X(Z) + \frac{1}{2}Z^{10}X(Z)$$

If the second was a half-strength *implosion,* the sign is just reversed,

$$Y(Z) = X(Z) - \frac{1}{2}Z^{10}X(Z)$$

If the second signal came in overlapping the first (say, at time 4), the numbers in the series are a bit messy-looking,

$y_t = (10, 5, 0, 4, -8, -8.5, 0, -2, 1.5, 3, 0, \ldots)$

but, rest assured, this series can be written out as having the transform,

$$Y(Z) = X(Z) - \frac{1}{2}Z^4\,X(Z) \qquad (4)$$

$$= (1 - \frac{1}{2}Z^4)\,X(Z) \qquad (5)$$

A Forth function to do this kind of factorization is shown in Listing One. This code has been contributed to the Forth Scientific Library and is awaiting review for its acceptance. (Volunteers to do code reviews are eagerly accepted!)

In general, we can do this for an arbitrarily complicated sequence of explosions and implosions, as long as the response of the earth remains linear (an example of a *nonlinear* response would be the state of the ground within a mile or so of a nuclear explosion). So, suppose we have a sequence consisting of an explosion at $t = 0$, a half-strength implosion at time $t = 2$, and a quarter-strength explosion at time $t = 3$,

$$B(Z) = 1 - \frac{1}{2}Z^2 + \frac{1}{4}Z^3 \qquad (6)$$

then,

$$Y(Z) = B(Z)\,X(Z) \qquad (7)$$

Now, without doing any integrals and just manipulating polynomials, we have done the *convolution theorem*,

$$y_t = \int x(t-\tau)b(\tau)d\tau \qquad (8)$$

Both equations (7) and (8) state that the signal y_t is the convolution of x_t and b_t, but the Z transform version just involves simple manipulation of polynomials. Listing Two provides an implementation of convolution algorithms; this code is also an unreviewed contribution to the Forth Scientific Library.

Convolutions and Digital Filters

Having established the convolution theorem is very important, since digital filters can be described in terms of convolutions. The Z transform version allows us to work with these filters by merely manipulating polynomials. In order to see that this is so, I have to reveal one neat fact about the Z transform. The Z transform we describe above can be written as,

$$X(Z) = \sum_t x_t Z^t \qquad (9)$$

If this is evaluated on the unit circle in the complex plane, that is we let $Z = e^{j\omega}$, we get

$$X(Z) = \sum_t x_t e^{j\omega\, t} \qquad (10)$$

which is the definition of the (discrete) Fourier transform! In other words, the Fourier transform is just the Z transform evaluated on the unit circle.

One way to interpret (7) is that we can make y be a series that has a spectrum that is the spectrum of x modified by the spectrum of b. So, for example, if we design b to have a zero at 60 Hz, y will be the signal x with 60 Hz removed, i.e., we have a notch filter.

In general, we can write a digital filter in the form,

$$Y(Z)\,A(Z) = X(Z)\,B(Z) \qquad (11)$$

or, written out in the time domain,

$$\sum_{k=0} y_{t-k}a_k = \sum_{k=0} x_{t-k}b_k \qquad (12)$$

Listing One.

```
\ coefdet     coefficient determination           ACM Algorithm #131

\ This routine takes two sets of power series,
\    H(x) and G(x), coefficients in the form:
\ H(x) = \sum_i=0^N h[i] x^i
\ and returns the power series which is the expansion of the quotient
\ H(x) / G(x)

\ The result is returned in H.
\ It is assumed that g[0] is nonzero.

\ This is an ANS Forth program requiring:
\       1. The Floating-Point word set
\       2. Uses words 'Private:', 'Public:' and 'Reset_Search_Order'
\          to control the visibility of internal code.
\       3. The immediate word '&' to get word addresses and '&!' to
\          alias arrays to 'DARRAY'.
\       5. Uses the words 'FLOAT' and 'DARRAY' to create floating
\          point arrays
\       6. The compilation of the test code is controlled by the
\          VALUE TEST-CODE? and the conditional compilation words in
\          the Programming-Tools wordset

\ Collected Algorithms from ACM, Volume 1 Algorithms 1-220,
\ 1980; Association for Computing Machinery Inc., New York.
\ ISBN 0-89791-017-6
\ (Note: the original publication had indexing errors, these are
\         corrected in this code).
\  (c) Copyright 1994 Everett F. Carter.  Permission is granted by the author to
\  use this software for any application provided this copyright notice is
\  preserved.
\ ================================================================================

S" /usr/local/lib/forth/fsl-util.fth" INCLUDED

TRUE TO TEST-CODE?

\ ================================================================================

CR .( COEFDET            V1.3            20 August 1994   EFC )

Private:

FLOAT DArray g{
FLOAT DArray h{


Public:

: coefdet ( &h &g n -- )
      >R
      & g{ &!             \ point to array g
      & h{ &!             \ point to array h
      R>

      1.0E0 g{ 0 } F@  F/                           \ calculate alpha

      DUP 1- 0 DO
                FDUP h{ i } F@ F*  -1.0E0 F*        \ calculate beta

                \ adjust the later H values
                DUP I 1+ DO
                             FDUP g{ I J - } F@ F*

                             h{ I } DUP F@ F+
                             F!
                          LOOP

                FDROP
            LOOP

      \ scale H by alpha
      0 DO  FDUP h{ I } DUP F@ F*  F! LOOP

      FDROP
;

Reset_Search_Order

TEST-CODE? [IF]   \ test code ==================================================

6 FLOAT ARRAY a{
6 FLOAT ARRAY b{

: ctest1_setup ( -- )

     6 0 DO 0.0E0 a{ I } F!  0.0E0 b{ I } F! LOOP
     2.0E0 a{ 0 } F!
     4.0E0 a{ 1 } F!
     7.5E0 a{ 2 } F!
     4.0E0 a{ 3 } F!
     4.5E0 a{ 4 } F!
     0.75E0 a{ 5 } F!

     1.0E0 b{ 0 } F!
     2.0E0 b{ 1 } F!
     3.0E0 b{ 2 } F!
     0.5E0 b{ 3 } F!
;

: ctest2_setup ( -- )
        6 0 DO 0.0E0 a{ I } F!  0.0E0 b{ I } F! LOOP

        6.0E0 a{ 0 } F!
        4.5E0 a{ 2 } F!
        1.5E0 a{ 3 } F!
        1.125E0 a{ 4 } F!

        3.0E0 b{ 0 } F!
        1.5E0 b{ 1 } F!
        0.75E0 b{ 2 } F!
;

: coef-test ( -- )

   ctest1_setup

   CR
   ." A:  "  6 a{ }fprint CR
   ." B:  "  6 b{ }fprint CR

  a{ b{ 6 coefdet

   ." A/B (should be 2 0 1.5 0 0 0) :  "    6 a{ }fprint CR

   ctest2_setup

   CR
   ." A:  "  5 a{ }fprint CR
   ." B:  "  5 b{ }fprint CR

   a{  b{ 5 coefdet

   ." A/B (should be 2.0 -1.0 1.5 0 0) : "   5 a{ }fprint CR
;

[THEN]
```

(Code continues on next page.)

If x represents the measurement data, and a and b are the (somehow) known filter coefficients, the above equation can be reorganized in a way that is more obviously useful,

$$y_t = \frac{1}{a_0}\left(\sum_{k=0} x_{t-k}b_k - \sum_{k=1} y_{t-k}a_k\right) \qquad (13)$$

This equation is a general formulation of a digital filter. If the a coefficients beyond a_0 are non-zero, it is called an Infinite Impulse Response (IIR) filter, because an input series x consisting of all zeros except at one point will result in an output series y that will always contain some amount of the original input (for real computers with finite *CELL* sizes, the output will actually eventually decay away because of truncation effects). An example IIR filter is what is known as the *leaky integrator*,

$$y_t = x_t + ay_{t-1}$$

For the special case where the a coefficients beyond a_0 are all zero, the filter is called a Finite Impulse Response (FIR) filter, because an impulsive input will cause an output that will eventually settle down to a steady state. A simple example is the *running average*,

$$y_t = \sum_{k=0}^{M} b_k x(t-k)$$

Everything one can know about a digital filter is contained in the ratio of the elements of A and B,

$$H(Z) = \frac{B(Z)}{A(Z)} \qquad (14)$$

Listing Two.

```
\  convolve             Linear and Circular Convolution of arrays

\ Calculates the convolution of two arrays.
\ Linear convolution of A{0..m-1} and B{0..n-1}
\  z{j} = Sum_i=0^{m+n-1} a{i} * b{j-i},
\   a{} and b{} are taken to be zero outside the range of their indices

\ Circular convolution of A{0..n-1} and B{0..n-1}
\  z{j} = Sum_i=0^{n-1} a{i} * b{j-i},
\   a{} and b{} are periodically repeated outside the range of their indices

\ This code conforms with ANS requiring:
\      1. The Floating-Point word set
\      2. Uses words 'Private:', 'Public:' and 'Reset_Search_Order' to control
\         the visibility of internal code.
\      3. The immediate word '&' to get word addresses
\      4. The word DArray to define Array pointers and the word &! to set them.

\ This algorithm is described in many places, see e.g.
\     Burrus, C.S. and T.W. Parks, 1985; DFT/FFT and Convolution
\     Algorithms, Theory and Implementation, John Wiley & Sons,
\     New York pp. 232, ISBN 0-471-81932-8

\  (c) Copyright 1994 Everett F. Carter.  Permission is granted by the
\  author to use this software for any application provided this
\  copyright notice is preserved.

CR .( CONVOLVE          V1.0          18 August 1994   EFC )

\ ==================================================================
S" /usr/local/lib/forth/fsl-util.fth" INCLUDED

TRUE TO TEST-CODE?

\ ==================================================================
Private:

FLOAT DArray a{
FLOAT DArray b{
FLOAT DArray z{
Public:

\ linear convolution
: convolve  ( &a m &b n &z -- , f: --  )          \ z{} = A{} * B{}
                                                   \ presumes that z{} is
      & z{ &!                                      \ big enough ( n+m-1 )
      SWAP & b{ &!
      ROT  & a{ &!

      OVER OVER + 1-
      0 DO
             0.0E0
             OVER I 1+ MIN 0 DO
                               J I - OVER < IF
                                              a{ I }    F@
                                              b{ J I - } F@
                                              F* F+
                                             THEN
             LOOP

             z{ I } F!
      LOOP
      DROP DROP
;

\ circular convolution,  note that ALL arrays have n elements
: circ_convolve ( &a &b n &z  -- , f: -- ) \ z{} = A{} (*) B{}
                                          \ presumes that z{} is big enough ( n )
      SWAP & b{ &!
      SWAP  & a{ &!

      DUP 0 DO
             0.0E0
             DUP SWAP 0 DO
                    a{ I } F@
                    J I - DUP 0< IF OVER + THEN

                    b{ SWAP } F@
                    F* F+

             LOOP

             z{ I } F!
      LOOP
      DROP
;

Reset_Search_Order

TEST-CODE? [IF]    \ test code =======================================
5 FLOAT Array f{
4 FLOAT Array g{
7 FLOAT Array z{
```

(Code continues on next page.)

```
: ctest_init1 ( -- )
        2.0E0 f{ 0 } F!
        2.0E0 f{ 1 } F!
        3.0E0 f{ 2 } F!
        3.0E0 f{ 3 } F!
        4.0E0 f{ 4 } F!

        1.0E0 g{ 0 } F!
        1.0E0 g{ 1 } F!
        2.0E0 g{ 2 } F!
        0.5E0 g{ 3 } F!
;

: ctest_init2 ( -- )
       1.0E0 f{ 0 } F!
       8.0E0 f{ 1 } F!
       1.0E0 f{ 2 } F!

       1.0E0 g{ 0 } F!
       2.0E0 g{ 1 } F!
       2.0E0 g{ 2 } F!
;

: cvolve_test1 ( -- )
    ctest_init1
```

Call For Papers

The European Forth Conference euroForth '97: Oxford, England

"Embedded Communications"

September 26-28 1997

euroForth, the annual European Forth Conference, will focus this year on the increasing use of communications within applications, ranging from embedded controllers connected to the outside world through modems, to Internet payment engines delivering secure cash transfers from domestic PCs, and including the process-control environment which connects peripherals as diverse as door access controllers, mass spectrometers, commercial laundry controllers, walking robots, and management systems.

All these and many more are the domain of modern Forth systems, and papers are sought on the following topics:

- Embedded networking, including Fieldbusses and embedded TCP/IP
- New development environments
- Portable software tools
- Virtual machines
- Formal methods
- Any Forth-related topic

Proceedings will be published for the conference, and will be available immediately after the conference from the Conference Organiser. They will also be published by the Forth Interest Group. There is a refereed section of the proceedings for those who desire peer review of their work.

Delegates from all parts of Europe and North America are expected. euroForth is a friendly conference at which time is made available for meeting people, for informal discussions, and for contacts.

euroForth '97 is being held in the lovely setting of St Anne's College, Oxford. St Anne's was founded in 1879 and was the first Oxford institution to offer University education to women. In 1979, it opened its doors to men as well. This college is within easy walking distance to the city centre, yet is free from the crowds of central Oxford.

For further information, please contact:
The Conference Organiser, EuroForth '97
c\o Microprocessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
England

tel: +44 1703 631441
fax: +44 1703 339691
net: mpe@mpeltd.demon.co.uk

```
     CR

     ." f(x) = "   5 f{ }fprint   CR
     ." g(x) = "   3 g{ }fprint   CR

     f{ 5  g{ 3  z{ convolve

     ." f*g  = "   7 z{ }fprint   CR
     ."  should be: 2 4 9 10 13 10 8 " CR
;

: cvolve_test2 ( -- )
     ctest_init2

     CR

     ." f(x) = "  3 f{ }fprint CR
     ." g(x) = "  3 g{ }fprint CR

     f{ 3  g{ 3  z{ convolve


     ." f*g = " 5 z{ }fprint CR
     ." should be: 1 10 19 18 2 " CR
;

: cirvolve_test1 ( -- )
    ctest_init1

    CR

    ." f(x) = "  4 f{ }fprint   CR
       ." g(x) = "  4 g{ }fprint   CR

    f{ g{ 4  z{ circ_convolve


    ." f(*)g  = "  4 z{ }fprint   CR
    ."  should be: 12 11.5 10.5 11 " CR
;

: cirvolve_test2 ( -- )
     ctest_init2

     CR

     ." f(x) = "  3 f{ }fprint CR
     ." g(x) = "  3 g{ }fprint CR

     f{ g{ 3  z{ circ_convolve

     ." f(*)g = " 3 z{ }fprint CR
     ." should be: 19 12 19 " CR
;

: convolve_test ( -- )

     cvolve_test1
     cvolve_test2
;

: cconvolve_test ( -- )

     cirvolve_test1
     cirvolve_test2
;

[THEN]
```

which is known as the *filter characteristic.* Foremost is the *power transfer function,* which describes what fraction of energy at a given frequency in the input appears in the output,

$$P(Z) = H(Z)\, H^*(Z) \qquad (15)$$

where H^* is the *complex conjugate* (the sign of the imaginary part is inverted) of H.

In an earlier column, I pointed out that the power transfer function is only part of the story. One should also consider what the filter does to the phase of the input. In general, a filter's effect on the phase is frequency-dependent, so there is a *phase transfer function,*

$$\Phi(Z) = -\tan^{-1}\frac{Im(H(Z))}{Re(H(Z))} \qquad (16)$$

The filter can also have an effect on *packets* of waves. These packets are what you get, for example, when you modulate one frequency by another. The frequency of the modulation envelope itself becomes a filterable component with its own transfer function, the *group transfer function,*

$$G(Z) = -\frac{dH(Z)}{dZ} \qquad (17)$$

All of these transfer functions are conventionally computed on the unit complex circle.

The characteristic function for the leaky integrator is,

$$H(Z) = \frac{1}{1 - aZ}$$

Converting this to the time domain gives the impulse response of the filter,

$$h_t = \left(\frac{1}{a}\right)^t \text{ for } t > 0, \text{ 0 otherwise}$$

For the running average filter, the function is,

$$H(Z) = \sum_{k=0}^{M} b_k Z^k$$

and its impulse response is

$h_t = b_t$ for $t = 0 \ldots M$, 0 otherwise

Certain forms for the polynomials A and B have proven to be useful and are widely used. For example, the bandpass Butterworth filter has the coefficients in the form,

$$B(Z) = b(Z^4 - 2Z^2 + 1) \qquad (18)$$

$$A(Z) = Z^4 + a_3Z^3 + a_2Z^2 + a_1Z + a_0$$

The Butterworth filter chooses to optimize the flatness of the bandpass region of $P(Z)$ and, in doing so, gives up simple forms for Φ and G.

Chebyshev filters are designed to give the sharpest possible transition between the bandpass and bandstop regions. The filter gets its name because the filter characteristic contains a special polynomial known as a Chebyshev polynomial, T_n.

$$P(j\omega) = \frac{1}{1 + \varepsilon^2 T_n^2(\omega)} \qquad (19)$$

This filter gains in the sharp transition by compromising on the flatness of the bandpass region.

Elliptical or Cauer filters get even sharper transitions than Chebyshev filters, but achieve this at the expense of ripples in *both* the pass and stop bands. The filter characteristic looks like the Chebyshev filter, except the Chebyshev polynomial gets replaced by a Chebyshev rational function.

Bessel filters are designed with the goal of achieving the flattest possible group delay. This results in a filter that has no ringing when it receives a step or impulse input. The characteristic function for the *n*th order Bessel filter is given by,

$$H(j\omega) = \frac{b_0}{q_n(s)} \qquad (20)$$

where,

$$q_n(s) = \sum_{k=1}^{n} b_k s^k \qquad (21)$$

$$b_k = \frac{(2n-k)!}{2^{n-k}k!(n-k)!}$$

Conclusion

We have barely scratched the surface on digital filters here. Go to any technical bookstore and you can easily find five hundred page books on the subject, so don't be surprised that I left something out. I am listing a few literature references to provide starting points on other places to look in order to learn more. In this installment, I have concentrated on describing the form of digital filters and not much upon how to design them. However, from what's been presented so far, it's pretty clear that designing a digital filter involves the manipulation of *A(Z)* and *B(Z)* in order to get the desired effect in *H(Z).*

Those of you who asked for me to do an article on *adaptive PID* will eventually get their wish. But before we can properly understand that topic, we first need to build a foundation by learning about digital filters, and then adaptive digital filters. With this column we have taken the first step with our first look at non-adaptive filters.

Please don't hesitate to contact me through *Forth Dimensions* or via e-mail at *skip@taygeta.com* if you have any comments or suggestion about this or any other ForthWare column.

References

Bracewell, R.N., 1986; *The Fourier Transform and its Applications,* McGraw Hill, New York, 474 pages, ISBN 0-07-007015-6

Claerbout, J.F., 1985; *Fundamentals of Geophysical Data Processing with Applications to Petroleum Prospecting,* Blackwell Scientific Publications, Palo Alto CA, 274 pages, ISBN 0-86542-305-9

Oppenheim, A.V, and R.W. Schafer, 1975; *Digital Signal Processing,* Prentice Hall, Englewood Cliffs NJ, 586 pages, ISBN 0-13-214635-5

Robinson, E.A. and S. Treitel, 1980; Geophysical Signal Analysis, Prentice Hall, Englewood Cliffs NJ, 466 pages, ISBN 0-13-352658-5

Rorabaugh, C.B., 1993; *Digital Filter Designer's Handbook, featuring C routines,* McGraw-Hill, New York, 332 pages, ISBN 0-07-053661-9

Skip Carter is a scientific and software consultant. He is the leader of the Forth Scientific Library project, and maintains the system *taygeta* on the Internet. He is also the President of the Forth Interest Group.