he next
orth generation
as syntax that
llows object-like
multiple code
ields

Part 1

Forth Shifts Gears

By George W. Shaw

The language Forth differs from other computer languages
n many respects. For example, the usual
rocess of defining data structures in
orth is to extend the compiler's knowl-
dge to include the new type. In contrast,
 allows *typedef* and *struct* to define the
ements of a data structure and then use
e named definition to create instances
f the structure.

A more specific example can be seen
n the definition of a *NUM*. In C, this
efinition might be:

```
pedef struct


    int n;
   num ;
um ten = { 10 } ;
n.n       /* access num in structure */
```

ompare this to the Forth definition:

```
 IUM   ( n - )( - adr )   ( create storage
  for n, returning pointer to it, adr)
  CREATE , ( n) ;
) NUM TEN
EN @    ( access num)
```

Both definitions allow access to
UM's storage. The entire definition
om *CREATE* to the semicolon (;) is
pproximately equivalent to the *typedef*
truct combination, plus any initializa-
on for each instance.

If, however, we wanted the structure to
e stored in external memory, neither *struct* nor *typedef* would be of much help. We would need to write a separate procedure to allocate the external memory to the new structure and manipulate the structure with procedure calls. In Forth, this procedure can be bundled into the defining word, making the process much cleaner:

```
: XNUM   ( n - )( - adr )
     ( create external storage for n,
         return ptr )
   CREATE   ..allot external memory,
     leave addr..
   DUP , ( save location)
! ( store init value)
DOES> @ ( external location ptr) ;
10 XNUM XTEN
XTEN @X   ( access xnum)
```

Note that the Forth code from *CREATE* to just before *DOES>* is approximately equivalent to the C *typedef struct* combination plus any initialization of each instance. This create-time code is executed during the creation of an instance of a *XNUM*. The code from *DOES>* to the ; (the does-time code) is executed when an instance of an *XNUM* is referenced. Conventional Forth memory reference syntax can be used to read and write the *XNUM*. The greater flexibility of Forth comes in the control over the create-time and does-time code.

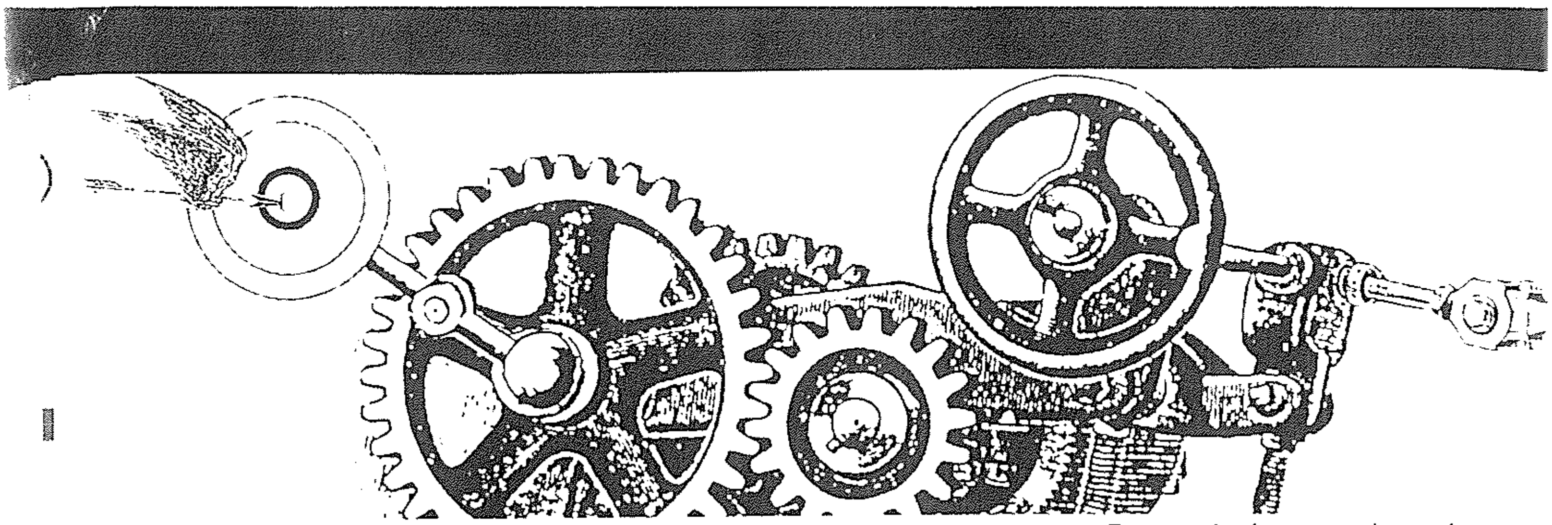
The preceding example is trivial yet representative. It is important to note that the code could be changed so the data is stored on disk or on a network. To do the equivalent in C would be exceedingly difficult (or impossible) while retaining C's *struct*-compatible syntax.

Unfortunately, each new class of Forth structures may require its own set of manipulation operators, cluttering the language with operator names surrounding a given theme. The example already mentioned might need the external memory read and write operators (named, say, *@X* and *!X*) once the external address is available. Many Forth systems have most or all members of the memory and I/O reading set (*@*, *2@*, *B@*, *C@*, *D@*, *N@*, *P@*, *PC@* and *S@*) as well as the memory and I/O storing set (*!*, *2!*, *B!*, *C!*, *D!*, *N!*, *P!*, *PC!* and *S!*) (Table 1).

Additionally, after almost every use of a data structure, an operator is compiled to define the action upon that structure. Considering the proliferation of operators, this is sometimes confusing, somewhat error prone, and almost always a waste of memory.

As the number of data structure classes (objects) increases, it is apparent that the number of operators (methods) also increases. This correlation makes it difficult to recall what operators are available for each structure, how they are spelled, and even how they are used.

Since these operators are defined separately, it usually is not possible to ask the computer what operators are available for a data structure without first creating and maintaining another data structure. However, removing the punctuation character or other attached prefixes or suffixes and using a different syntax might make the operators easier to use and the code more readable.

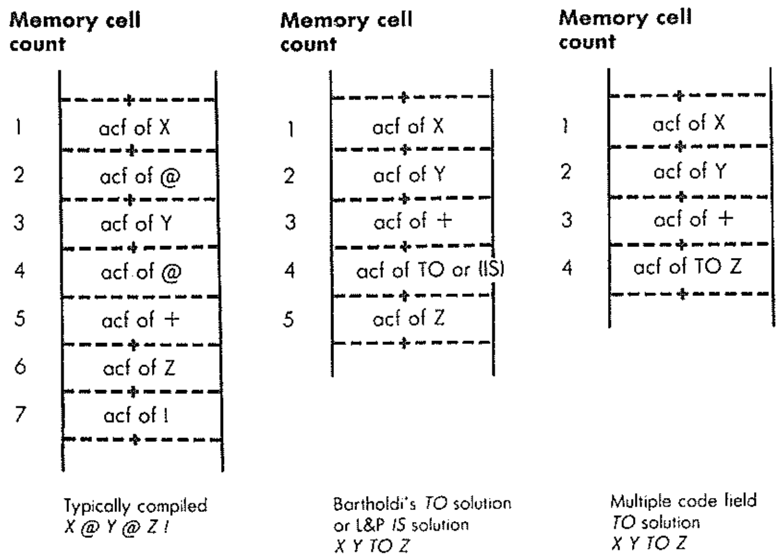


Memory and I/O reference operator set

@	Read 16 bits from memory
2@	Read 32 bits from memory
B@	Read 8 bits from a record in the disk buffers
C@	Read 8 bits (character) from memory
D@	Read 32 bits (double) from a record in the disk buffers (sometimes an alias for 2@)
N@	Read 16 bits (number) from a record in the disk buffers
P@	Read 16 bits from an I/O port
PC@	Read 8 bits from an I/O port
RP@	Read return stack pointer
S@	Read string into PAD buffer
SP@	Read data stack pointer
!	Store 16 bits into memory
2!	Store 32 bits into memory
B!	Store 8 bits into a record in the disk buffers
C!	Store 8 bits (character) into memory
D!	Store 32 bits (double) into a record in the disk buffers (sometimes an alias for 2!)
N!	Store 16 bits (number) into a record in the disk buffers
P!	Store 16 bits into an I/O port
PC!	Store 8 bits into an I/O port
RP!	Store into return stack pointer
S!	Store string into PAD buffer
SP!	Store into data stack pointer

Table 1.

Compiled representations



Typically compiled
X @ Y @ Z !

Bartholdi's *TO* solution
or L&P *IS* solution
X Y TO Z
X Y IS Z

Multiple code field
TO solution
X Y TO Z

Typical direct or indirect threaded code implementations
acf = address of code field

Figure 1.

For example, the command to read memory is @ (called "fetch") and the command to write memory is ! (called "store"). Thus the command to read the clock is often *@TIME* ("fetch time") and the command to set the clock is often *!TIME* ("store time"). The meaning of these commands is readily apparent to most Forth programmers without ever looking them up in the reference manual. But are these the correct spellings? Could they alternatively be *TIME@* and *TIME!*? Certainly. And, if one exists, what of the word set to operate the serial port? Is it *COM1@* and *COM1!* or *@COM1* and *!COM1*? Each of these examples requires two identifiers to name the two operations.

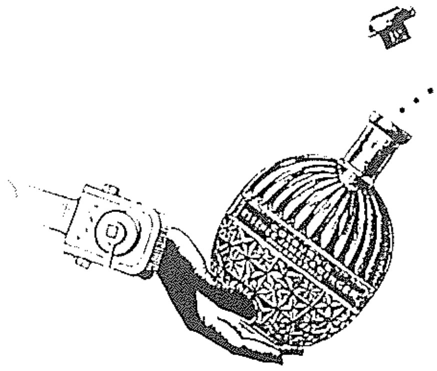
Application examples are often complex, requiring many more operators. It might also be possible with a new syntax to perform these and other operations with a single identifier and perform operations not conveniently possible without a new syntax.

The advent of such a syntax allows for previously unforeseen possibilities in Forth. You can define generic smart operators that require no smarts at all because the intelligence is built into the data structures. Forth thus possesses capabilities much like object-oriented languages, with the ability to manipulate many varied data structures using a few simple logical operators. Additionally, the run-time references compile to less memory and execute faster than the previous dumb operator-reference pairs.

Parallel histories

Over time, Forth has developed as a generalized solution to what were once specific problems. The same is true of the development of the object-oriented programming techniques described in this article, though object-oriented programming was never the goal.

The seed for this work came from the growing dissatisfaction of Forth users with operators, stemming from constructs like *X @ Y @ + Z !*. This structure obtains the values of the variables *X* and *Y*, adds them, and stores the result in *Z*. Referencing the variables returns a pointer to their data. @ and ! read or set memory, given the pointers.



Note that since in Forth all identifiers are separated by spaces, each is generically referred to as a word. The seven words in the previous example compile to seven cells in memory. Further, analysis has shown that values are fetched many more times than they are stored. Thus the @ following *X* and *Y* is usually redundant, requiring additional memory and execution time.

These problems were first discussed by Charles Moore in 1978.[1] At that time, Moore also suggested the syntax for what was later to become known as the *TO* solution *X Y + TO Z*. The concept involved the variables *X* and *Y* normally returning their values rather than pointers to their values, and the *TO* prefix causing the result to be stored in *Z*. The primary goal was to eliminate the required pointer manipulation (so that standardizing pointers would become a simpler task) and create a more readable and more efficient syntax. This new syntax typically compiles to four or five cells (two to three cells less than the original) and can execute much faster, depending on the implementation (Figure 1).

Several implementations of this syntax have been presented since its introduction. Both run-time bound[1] and compile-time bound[2,3] implementations have been presented. Various extensions such as *AT*, *OF*, and *+TO* have added more capabilities and extended the syntax. Alternatives such as the *IS* solution solve the original, limited problem more simply, but with contracted additional capabilities.

The words *QUAN* (a variable/constant type word), *LQUAN* (a variable/constant type word in external memory), and *VECT* (a vectored execution variable) were suggested to define structures using the *TO* solution syntax. These names have lost favor and been replaced by the more appropriate and readable *VALUE* and *DEFER*.

Unfortunately, little other work has been presented that fully realizes the power of all these implementations. The object-oriented programming techniques presented in this article are an attempt to generalize the *TO* solution and its various implementations. (For a more thorough discussion of the history of the *TO* solution, see the accompanying sidebar.)

History of the *TO* solution

The *TO* solution was originally presented by Charles Moore at the Forth standardization meeting in 1978. He suggested the following syntax:

```
X Y + TO Z
```

to resolve Forth programmers' dissatisfaction with operators.

Moore's suggestion was implemented by Paul Bartholdi.[1] This implementation caused a run-time variable, set by *TO*, to indicate the action to be performed by the next variable, *Z*. Variables would otherwise return their value (unless preceded by *TO*) when they set their value. This solution was used widely in Europe and especially at the University of Utrecht, the Netherlands.

Bartholdi's implementation has several limitations. First, it uses run-time binding (a conditional test is executed at run time) that slows execution. Second, it does not allow easy access to a pointer to the variable (the value previously returned by variables). Third, the Forth operator *+!*, which increments the value of a variable given its pointer (similar to +=, ++, −=, and -- in C) cannot be used.

Various extensions were proposed, all requiring even more complex conditional run times:

```
AT Z (return the pointer to Z)
+TO Z (add to the value of Z)
```

The complexity and speed of the run-time binding are even more unsatisfactory considering other operations on *Z* might be defined.

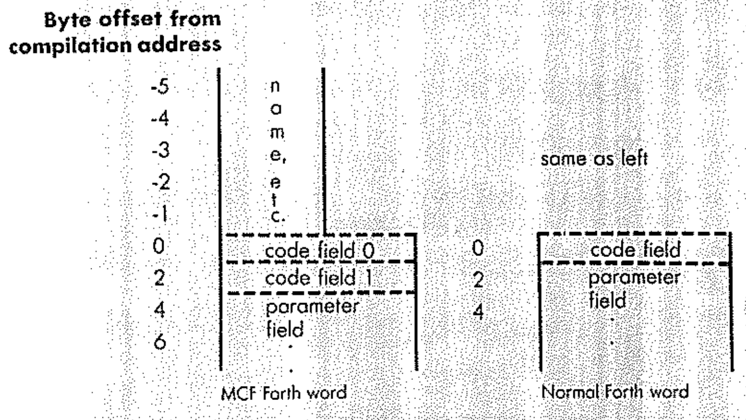
George Lyons suggested a more elegant solution just after Bartholdi published his *TO* solution.[2] Lyons suggested that a variable might have two run-time operations rather than one, the first for the fetch and the second for the store operation. The compiler or interpreter could then decide which to use. Lyons' solution did not allow for returning a pointer to the variable but did solve the run-time speed and complexity problems by binding the operation at compile time.

The run-time code for a Forth word is located through the code field of the compiled word definition. The code field contains a pointer to the run-time code (indirect threaded code) or the actual code itself (direct threaded code), depending upon the implementation. Thus, Lyons proposed that variables have two code fields to locate the two pieces of run-time code (Figure 1). Other implementation techniques use other combinations of run-time code and pointers, but the concepts remain the same.

Evan Rosen presented his implementation of Lyons' solution at the 1982 Forth Modification Laboratory (FORML) conference.[3] He used three code fields, one each for fetching, storing, and pointing. These types of structures have come to be known as multiple code field (MCF) words.

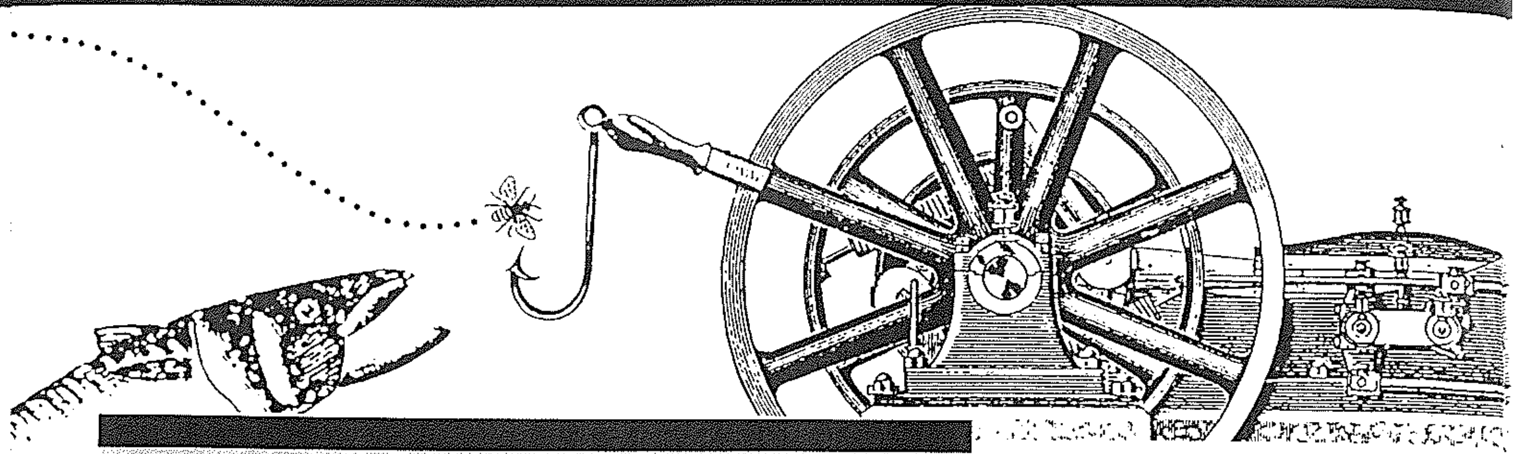
Rosen named his MCF variable *QUAN*. He also created an execution vector (an executable pointer to a procedure) he named *VECT*. Unfortunately, he did not devise a syntax

Lyons' MCF word memory layout



In either configuration, the value of the *VALUE* or the procedure assigned to the *DEFER*ed word is stored directly in the parameter field of the word.

Figure 1.



for easily defining classes of MCF words. Each class-defining word was constructed by hand as a combination of high-level defining and assembly code run times. Further, his *TO* and *AT* words were created as state smart (this term is discussed at length in the main body of the article).

Despite these limitations, several Forth vendors have implemented an *LQUAN* whose data exists in external (to Forth) memory. Since most Forth systems at the time used only 16-bit addresses, this greatly increased the address space for data available to the system.

Klaus Schleisiek generalized the defining sequence.[4] However, the syntax he used (adapted from William Ragsdale[5]) is not compatible with standard Forth syntax for defining words. His implementation allows new classes of words and new selectors to be easily defined and the run times for the new classes to be in both high-level Forth or assembly language code.

Henry Laxen and Mike Perry published a public domain Forth-83 implementation of Forth with *IS* as an alternative to *TO*. They used *CONSTANT* to define the variables and added the word *DEFER* to define execution vectors (rather than *VECT*).

Martin Tracy further enhanced the *IS* solution with the defining word *VALUE* to replace Laxen and Perry's nonstandard use of *CONSTANT*. A *VALUE* is defined to behave like a *CONSTANT*, normally returning its value. A *DEFER*ed word is defined similarly in structure, except that it executes the pointer to the definition assigned to it. *IS* can be used on both as it simply stores into the parameter field (data storage area) of the word following it. Though these are not MCF implementations, fetch operations on a *VALUE* require less memory and are fast, while store operations require the same amount of memory but are slower than before.

The *IS* solution solves the original complaint from Moore about pointer usage, syntax, speed, and memory requirements of variables. Like *DEFER*, *IS* can be used with 16-bit structures other than *VALUE*s. However, many Forth systems, especially multitasking user systems (a natural in Forth) have local variables. These are dynamic local or static user variables usually implemented as an offset from a local area pointer.

These systems typically also have local *DEFER*ed words, requiring Laxen and Perry to make *IS* smart to store into the different types of *VALUE*s and *DEFER*ed words properly. The other alternative was a dumb *IS* that would have been unusable on local structures. Further, *VALUE* and *IS* do not allow easy access to a structure's pointer or use of the *+!* operator, *+TO* type prefix, or data other than 16-bit.

For common use, I prefer the names *VALUE* and *DEFER* rather than *QUAN* and *VECT* because they are full words and more readable. Also, rather than using Rosen's suggested method selector *AT* to refer to the pointer to an instance of a *VALUE*, I prefer *OF* (as in "address of"). *AT* is easy to confuse conceptually with @, implying the contents of the address rather than the address itself. *AT* is also used in many systems, such as Laxen and Perry's public domain system, for *X Y* cursor addressing.

References

1. Bartholdi, Paul. "The *TO* Solution," *Forth Dimensions*, 1(4): 38-40, 1979.
2. Lyons, George. Letter to the editor, *Forth Dimensions*, 1(5): 56, 1979.
3. Rosen, Evan. "High Speed, Low Memory Consumption Structures," *1982 FORML Conference Proceedings*, pp. 191-196.
4. Schleisiek, Klaus. "Multiple Code Field Data Types and Prefix Operators," *The Journal of Forth Application and Research*, 1(2): 55-63, 1983.
5. Ragsdale, William. "A New Syntax For Defining Words," *1980 FORML Conference Proceedings*, pp. 122-130

What are MCFs?

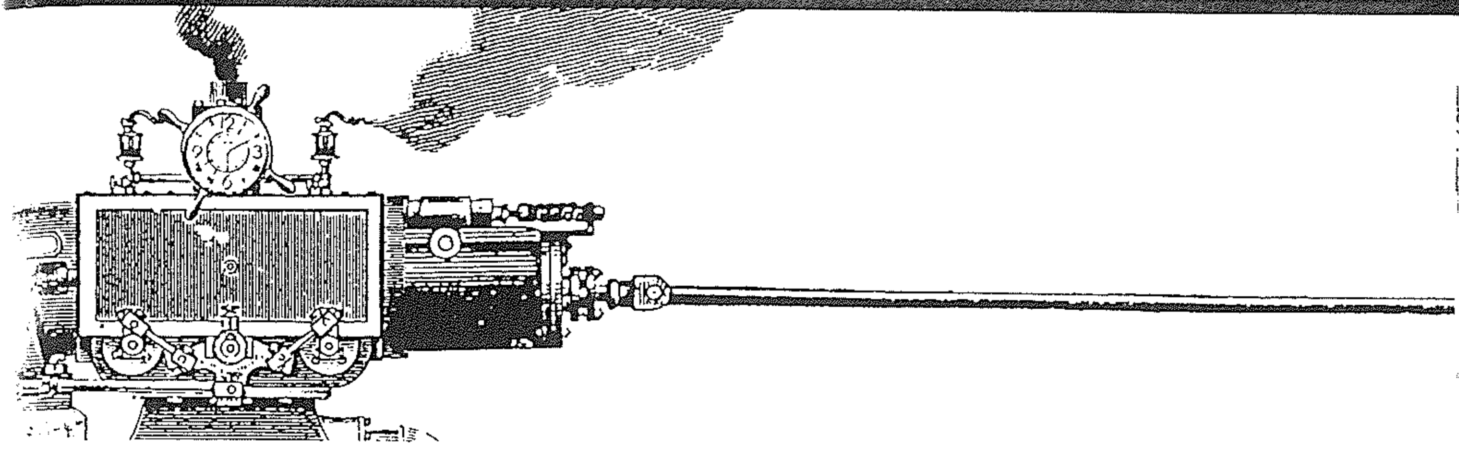
In the examples at the beginning of this article, the default action of *CREATE* or the code following *DOES>* specifies the run-time action of *TEN*. The system finds the run-time code for a Forth word through the code field of the compiled word definition. The code field contains a pointer to the run-time (indirect-threaded) code or the actual (direct-threaded) code itself, depending on the implementation.

Other implementation techniques use different combinations of run-time code and pointers, but the concepts remain the same. This single code field allows a single run-time behavior to be associated with the data. To efficiently allow multiple run-time behaviors, multiple code fields (MCFs) are needed. The concept of multiple rather than single behaviors (run times) is the focus of this article.

From the programmer's point of view, it is useful not to have to know how a data structure is organized to use it—somewhat like driving a car without knowing what type of engine it has or precisely how it works. Maintaining this arms-length distance between the working internals and the operations upon the data structure is known as "data abstraction." The operations defined to manipulate the data structures are "methods." (Methods can be any manipulation appropriate for an object, including adding, dividing, moving, storing, executing, and printing.)

The words that tell the system which method to use are "method selectors." What the methods manipulate and whose methods are selected are "objects." (Anything can be an object, including serial ports, clocks, procedures, numbers, and data structures. In Smalltalk, *everything* is an object.) The ability to define a new class of objects and associated methods created as an extension or modification of an existing class is "inheritance."

In object-oriented programming, objects are treated as atomic units to maintain data abstraction. Maintaining data abstraction allows the structure of the object to change during program development, program execution, or when moving from system to system without breaking the program. In keeping this



arms-length relationship, a method refers to a more logical rather than strictly physical operation upon an object. Meth-od identifiers are then easily and appropriately reused, greatly reducing the number of operations for the programmer to remember.

Data abstraction and shareable method selectors allow the programmer to write generalized routines to perform operations upon a wide variety of objects without care for their internal structure. mplemented properly, the methods execute very efficiently on their objects.

Object-oriented Forth

Forth has always had the buddings of an object-oriented language. For example, the standard *DOES>* construct used to create *VALUE* in effect allows the definition of a single method upon the object data structure *VALUE* (the first MCF word class):

```
VALUE       ( n - )( - n )
  ( define CONSTANT like structure )
    CREATE        ( n ) ,
    DOES>         @    ( n )        ;
```

What is needed is a syntax and mechanism to expand *DOES>* to allow it to define additional methods. MCFs give us the essence of object-oriented programming in Forth. Each code field specifies a method to be used on an instance of its object. Properly implemented, methods can be inherited.

TO, *AT*, and *+TO* (among others) are method selectors. By using these selectors rather than explicit operators, messages are passed through the compiler and interpreter and received by the desired object. Thus data abstraction is maintained. Further, the selectors are logical operations that can be applied to a wide range of structures.

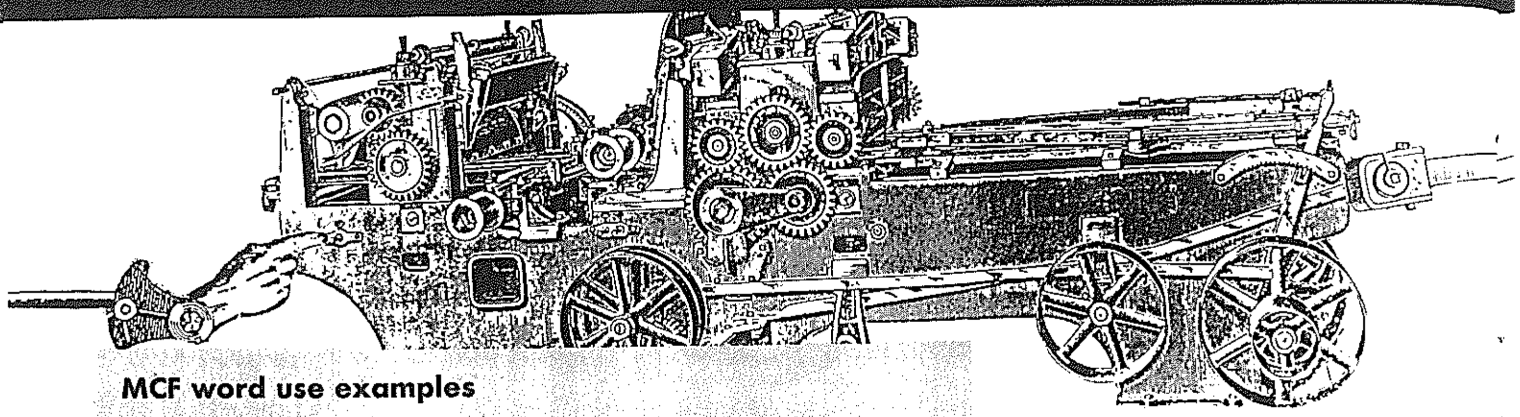
Listing 1 shows *VALUE* and *DEFER* defined using the proposed new syntax. *DOES>* returns a pointer to the data object. The *TO* and *OF* prefixes tell *DOES>* which method to apply to the procedure. This new syntax is a direct extension of the existing standard syntax and use of *DOES>*.

*VALUE*s and *DEFER*s are not the only uses for MCFs and object-oriented programming in Forth. As already mentioned, these structures were not originally intended to add objects to Forth but to clean syntax, speed execution, and save memory. Primarily, their intent was to reduce the use of pointers to enhance transportability.

A secondary effect of *LQUANS* is the ability to address external memory without explicit external addressing. By using MCFs to hide the external addressing mechanism, this extension further exercised the beginnings of object-oriented programming: data abstraction. Remember, one of the attributes of object-oriented programming is that the internals of a structure remain hidden unless explicitly asked for.

```
                    ( Method selectors )
: TO   2 TO CF-   ;   ( select code field 2 bytes back )
: OF   4 TO CF-   ;   ( select code field 6 bytes back )
: +TO  6 TO CF-   ;   ( select code field 4 bytes back )

                    ( Methods of VALUE )
: +TO-VALUE  +TO DOES>  +!  ;   (S n - )( increment value )
: OF-VALUE   OF DOES>       ;   (S - a )( return data address )
: TO-VALUE   TO DOES>   !   ;   (S n - )( store value )
: DO-VALUE   DOES>   @      ;   (S - n )( return data value )

: VALUE     (S n - / to: n - , of: - a , - n )
   HEADER   4 CODEFIELDS   ,   TO-VALUE +TO-VALUE OF-VALUE DO-VALUE   ;

Or define as...

: VALUE     (S n - / to: n - , of: - a , - n )
   HEADER   4 CODEFIELDS   ,   TO-VALUE +TO-VALUE OF-VALUE   DOES>   @   ;

Without naming DO-VALUE.  Similarly...

: DEFER     (S cf - / to: cf - , of: - a , ? - ? )
   HEADER   3 CODEFIELDS   COMPILE NOOP   TO-VALUE   OF-VALUE
   DOES>   @ EXECUTE   ;   ( note reusability of named run-times )

Or use inheritance and define as...

: DEFER     (S cf - / to: cf - , of: - a , ? - ? )
   ['] NOOP VALUE   DOES>   @ EXECUTE   ;
( above will contain inappropriate +TO function )

Examples:

100 VALUE SCALE       ( define to initially return 100 )
1000 TO SCALE         ( set to return 1000 )
SCALE                 ( return value )
-2 +TO SCALE          ( increment value by -2 )
OF SCALE              ( return pointer to data )

DEFER KEY             ( define vector to character input )
' (KEY) TO KEY        ( set KEY to execute (KEY) )
KEY                   ( return next keypress from keyboard )
OF KEY                ( return pointer to current routine )
```

Listing 1.



MCF word use examples

VALUE			
USER VALUE			
2VALUE			
USER 2VALUE			
	VALUE <name>	(n -	Defines name for value n)
	<name>	(- n	Returns value n)
	TO <name>	(n -	Set value to n)
	OF <name>	(- a	Return address of n)
	+TO <name>	(n -	Increment <name> by n)
Reduces usage of @ and ! and speeds execution.			
DEFER			
USER DEFER			
	DEFER <name>	(-	Defines name to be deferred)
	<name>	(?	Executes assigned procedure)
	TO <name>	(a -	Sets name to execute word at a)
Eliminates *IS*, runs faster, and requires less memory. *DEFER* can be extended to allow MCF words to be deferred and manipulated.			
SP			
	SP	(- a	Returns stack pointer value)
	TO SP	(a -	Set stack pointer value)
Eliminates the need for *SP@* and *SP!*			
RP			
	RP	(- a	Returns return stack pointer value)
	TO RP	(a -	Set return stack pointer value)
Eliminates the need for *RP@* and *RP!*			
STRING			
	STRING <name>	(n -	Define string storage)
	TO <name>	(a # -	Set string value)
	OF <name>	(- a #	Return maximum string)
	<name>	(- a #	Return current string)
Eliminates the need for *S@*, *S!*, and *MAXLEN*.			
CASELESS			
	TRUE TO CASELESS		(Set no case dictionary search)
	FALSE TO CASELESS		(Set case-sensitive dictionary search)
	CASELESS	(a # - a #	Convert string to uppercase if set)
Eliminates need for separate variable to hold flag and run-time conditional test			
HERE			
	HERE		(Returns dictionary pointer)
	TO HERE		(Set dictionary pointer)
	+TO HERE		(Increment dictionary pointer)
Eliminates the need for *DP*, speeds execution, and can eliminate the need for *ALLOT* by using *+TO*.			
SINGLE			
DOUBLE			
BYTES			
TEXT			
	File fields of various types. Treated just like resident VALUEs or STRINGs. Record buffer resident or disk buffer resident data.		
Eliminates the need for disk buffer field fetch operators (*D@*, *N@*, *B@*, etc.) and store operators (*D!*, *N!*, *B!*, etc.) and allows byte-order independent data transferring.			
GROUP			
	GROUP <name>		(Defines a file field group)
	<name>		(Acts like a vocabulary to select fields)
	OF <name>	(- a #	String containing all subfields)
	TO <name>	(a # -	Set subfield group to sting)
Eliminates group fetch and store operators. Syntax difficult if conventionally defined. Similar to *struct* in C.			

Table 2.

MCFs and new syntax options

*DEFER*ed words were an obvious choice for the second MCF word class. *DEFER*ed words are very similar to *VALUE*s and inherit most of their behavior. In fact, as *VALUE* obviates most needs for *VARIABLE*s, *DEFER*ed words are implemented using *VARIABLE*s on systems not containing an appropriate defining word:

```
VARIABLE 'KEY        ( variable to hold
                     pointer to key
                     routine)
: (KEY) ...     ;    ( run-time key
                     routine )
' (KEY) 'KEY !       ( set vector to
                     pointer to key
                     routine )
KEY                  ( - c )( execute KEY
                     routine to get a
                     character )
    'KEY @ EXECUTE      ;
```

DEFER defines an execution vector (a forward reference) to a procedure to be later defined or changed during execution. With an MCF *DEFER* (as with an MCF *VALUE*), the method selector *TO* is used to set the procedure to be executed, completely eliminating the need for the word *IS* and its complexity (Figure 1). Additionally, memory is saved each time a *DEFER*ed word or *VALUE* is set because *IS* no longer compiles before *DEFER* or *VALUE*.

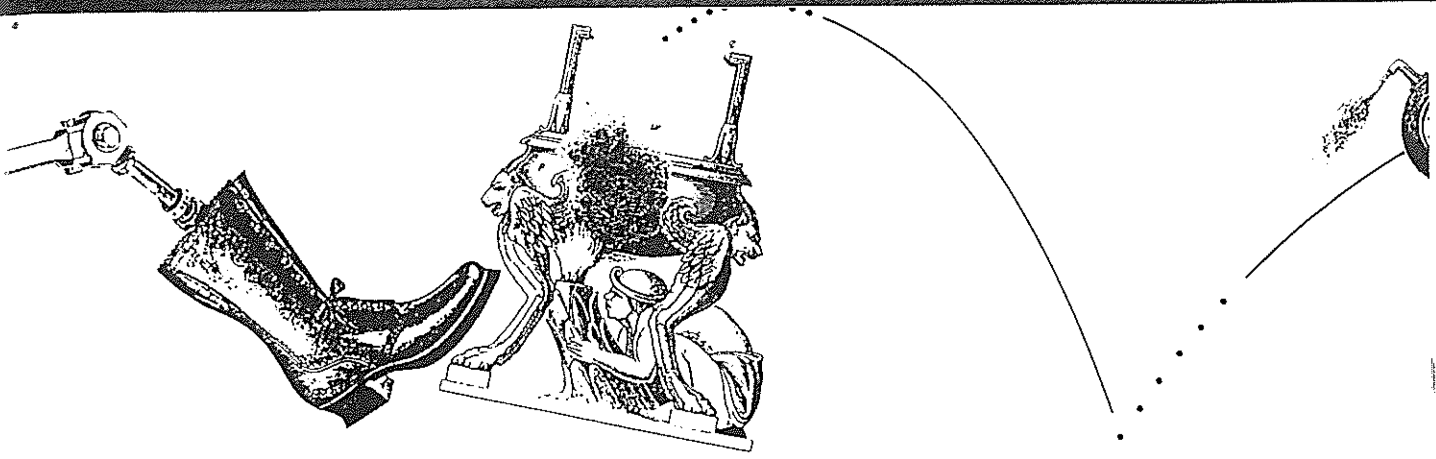
How else might we use these and other MCF words? For what other areas do we define a set of methods to work on an object? Consider the following clock example. As an MCF word, the two separate words with @ and ! prefixes or suffixes are no longer needed. Reading and setting the clock becomes:

```
TIME ( read the clock)
TO TIME ( set the clock)
```

Similarly, *DATE* can be used in the same way:

```
DATE ( read the calendar)
TO DATE ( set the calendar)
```

In addition, most Forth implementations include words to read and set the stack pointers—*@SP*, *!SP* or *SP@*, *SP!*



for the data stack and *@RP*, *!RP* or *RP@*, *RP!* for the return stack. Most also have string variables (often defined with *STRING* and accessed with *S@* and *S!*), the dictionary pointer (normally *HERE* to return its value, *DP* as the internal variable, and *ALLOT* to change its value while testing for dictionary full).

Table 2 lists the new syntax possible for these and other words. In addition to the memory saved when they are used, MCF words can save memory in the implementation of their methods, especially when the structures involved are very similar.

Most Forth systems have two stacks, a data stack and a return stack. Words exist to transfer data between the two stacks, but most operations are defined only on the data stack. Some Forth systems, though, supply a small subset of the most useful data stack words to operate on the return stack. These words usually have the same name as the data stack operation but are prefixed with an *R*. For instance, *SWAP* becomes *RSWAP*, and *DROP* becomes *RDROP*.

More operations are not commonly defined because the return stack is generally not used heavily; thus the memory usage is not justified. MCFs offer a very inexpensive mechanism for allowing all operations to be available on the return stack or even a user stack, if desired:

Define a method selector and call it *R*, which will select the second method. Define a method selector and call it *U*, which will select the third method. Consider the following syntax:

```
R SWAP
R OVER
R DROP
   (manipulate the return stack)

addr U SWAP
addr U OVER
addr U DROP
   (manipulate a user stack)
```

Now make all of the stack primitives MCF words. The second code field executes code (selected by *R*) that swaps the return stack and data stack pointers, executes the first code field (which performs the function on the current stack pointer), and then swaps the stack pointers back. This code need only be defined once with all the stack manipulation words pointing to it. Thus for the expense of about 20 bytes (plus 2-4 bytes per stack operator), all the data stack operations can be performed on the return stack.

For an additional 2-4 bytes per stack operator, a third code field (selected by *U*) can be added and passed a pointer to a user stack pointer. Manipulating the user stack is identical to manipulating the return stack except the user stack pointer and data stack pointer are swapped (Listing 2).

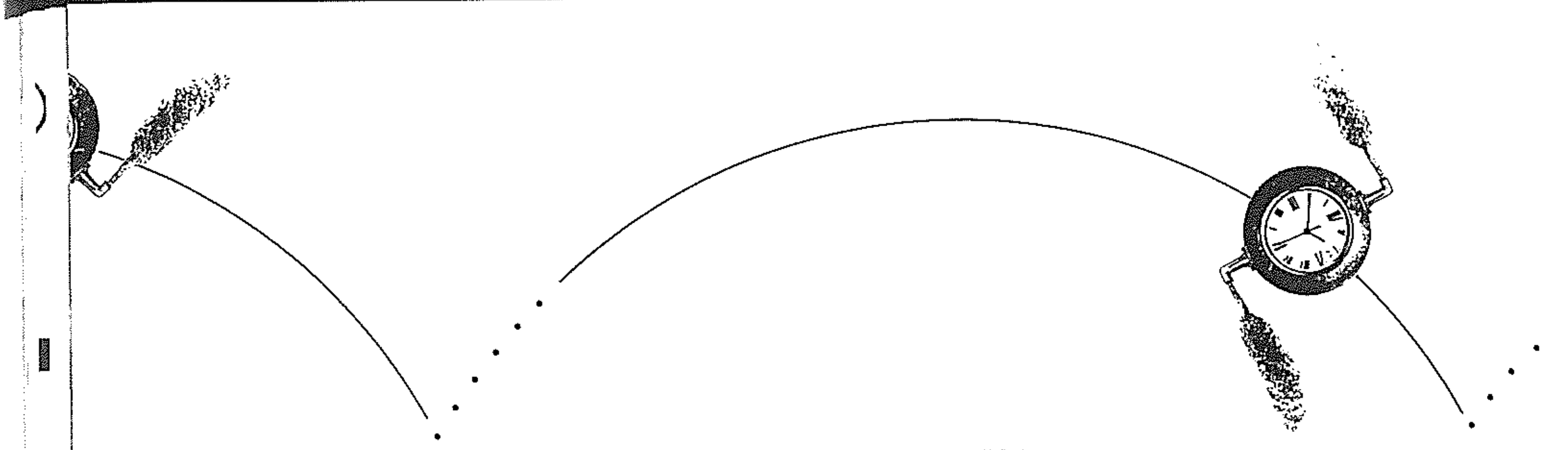
Further uses of MCF techniques and objects are unrelated to application data structures. Many systems allow the selection of case-sensitive or case-insensitive dictionary searching, usually performed by setting a flag tested during the dictionary search to control sensitivity testing. Testing flags is slow and inelegant. Using a *DEFER*ed word set for case or caseless dictionary searching is faster, but not as clean as it could be. It requires the user to remember (and the Forth to have in memory) the identifiers of the case-sensitive routine, the case-insensitive routine, and the *DEFER*ed word. Thus three identifiers are required to implement one operation.

A cleaner mechanism is to implement a single MCF word, *CASELESS*, that has two methods. *CASELESS* takes a string and, if necessary (depending on how the word was previously set) converts the string to uppercase. The *TO* method accepts a true or false flag and sets the word to convert to uppercase if necessary. Thus:

```
TRUE TO CASELESS
   (set to convert to uppercase)
FALSE TO CASELESS
   (set to not convert to uppercase)
```

```
LABEL RET-SWAP-OP   ( return here after return swap operation )
    HERE 2+ , ( code field) BP SP XCHG   SI POP   NEXT
LABEL RETURN-OP   ( swap stack pointers and do self operation )
    SI PUSH   BP SP XCHG   RET-SWAP-OP # SI MOV   2 [BX] JMP

LABEL USER-SWAP-OP   ( return here after user stack operation )
  HERE 2+ , ( code field) BP SP XCHG  DI POP  BP 0 [DI] MOV  BP POP
  SI POP   NEXT
LABEL USER-OP   ( swap stack pointers and do self operation )
    DI POP   SI PUSH   BP PUSH   DI PUSH   0 [DI] BP MOV
    SP BP XCHG   USER-SWAP-OP # SI MOV   4 [BX] JMP

SI = interpretive pointer, BP = return stack pointer,
BX = CF being executed

HEADER DROP   ( n - )( drop item from selected stack )
  3 CODEFIELDS        ( optimized )
    ;CODE   2 # SP ADD   NEXT   END-CODE
    R ;CODE   2 # BP ADD   NEXT   END-CODE
    U ;CODE   BX POP   2 # 0 [BX] ADD   NEXT   END-CODE

HEADER SWAP   ( n1 n2 - n2 n1 )( swap top two elements of stack )
  3 CODEFIELDS
    ;CODE   DX POP   AX POP   DX PUSH   AX PUSH   NEXT   END-CODE
    R ;USES RETURN-OP , ( swap pointers and do swap)
    U ;USES USER-OP ,   ( swap pointers and do swap)

Additional stack words are defined like DROP (optimized)
or more generally (but slower) like SWAP.
```

Listing 2.



A *FAR*-fetched idea

Common long operators and *FAR* counterparts:

C@L	FAR C@	addr page#—8b
@	FAR @	addr page#—16b
@L	FAR 2@	addr page#—32b
C!L	FAR C!	8b addr page#—
!L	FAR !	16b addr page#—
2!L	FAR 2!	32b addr page#—
CMOVEL	FAR CMOVE	addr1 page# addr2 page# count—
CMOVE>L	FAR CMOVE>	addr1 page# addr2 page# count—
FILLL	FAR FILL	addr page# count 8b—

Other operators which could be available in FAR versions:

SKIP	SKIP>	SCAN	SCAN>	SEARCH	COMPARE	ENCLOSE
CSET	CTOGGLE	CMASK	CRESET	ERASE	BLANK	INSERT
DELETE	REPLACE	CONCAT	APPEND			

For systems with external ram disk buffers:

FAR BLOCK FAR BUFFER

16-bit systems often have *L* suffixed operators to access addresses outside of their 64K data segment. Rather than duplicating names and adding on *L* suffix, a single MCF word can be used with the *FAR* operations as the second code field. The normal code field works as usual.

Table 3.

```
Forth allows the creation of words to control the compiler.  As an example of
the state-smart anomaly, we will create a renamed definition of [COMPILE].
[COMPILE] is used to force the compilation of compiler directives (words that
would otherwise direct the compiler to do something other than simply compile
themselves).  [COMPILE] is typically used when creating new compiler
directives and for other purposes.

: '         ( - a )( leave address of word following )
    BL WORD   FIND   0= IF ( not found error ) THEN
    STATE @ IF ( compiling) [COMPILE] LITERAL   THEN   ; IMMEDIATE

: [COMP]  ( - )( force compilation of next word )
    [COMPILE] '    ,    ;   IMMEDIATE

If [COMP] is used during compilation, STATE will be true and ' will compile
the address of the next word as a literal rather than allowing , (comma) to
compile it.
```

Listing 3.

Since *TRUE* and *FALSE* already exist in most systems, only *CASELESS* need be defined.

Far-fetched idea

Where else can we apply the MCF technology? Where else are operators replicated with slight changes in behavior to work on a related class of objects? A clear set—@ and !—was listed at the beginning of this article.

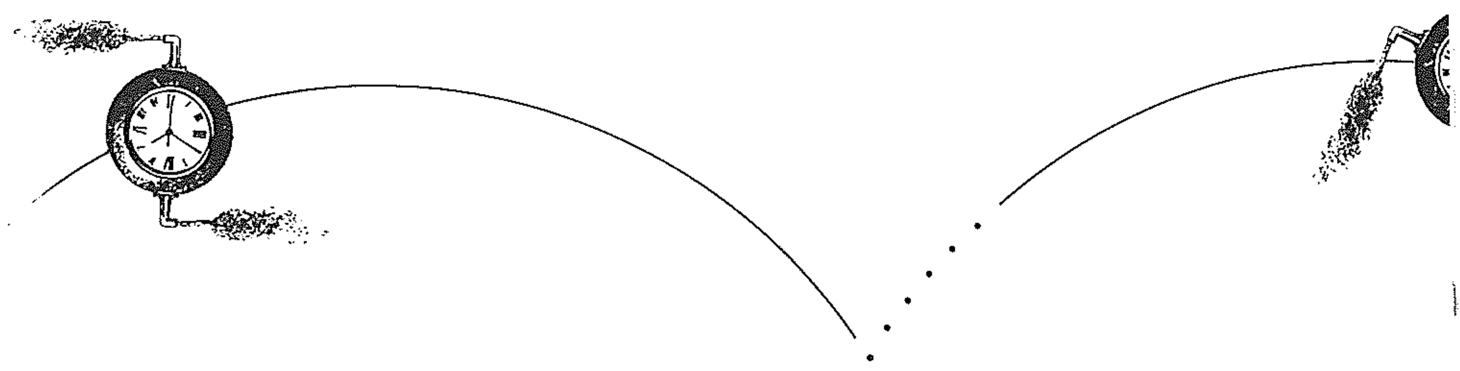
Many processors have 16-bit data busses. Unfortunately, they also have 20-bit, 24-bit, or larger address busses. Because the data bus width is less than the address bus width, CPUs such as the 8086, 80286, Z8000 and PDP-11 series are handicapped in working with addresses larger than their data bus width. For efficiency, most Forth systems for these processors are built around a 16-bit model with a 64K address space.

Operating Forth with a 32-bit model is one solution to this problem. However, this solution is inefficient in memory usage and processor time. Additionally, sometimes it may not be desirable to implement full 32-bit systems in 32-bit addressing machines (such as the 68000). Typically, 32-bit systems require twice as much program memory and, with their full linear addressing, do not supply the segment security automatically given by the former class machines.

The segmented machines typically use additional fetch and store operators to access this external memory. The external addressing word set often use an *L* suffix to denote long addressing. Thus @ becomes *@L* and *!* becomes *!L*. Double and character operators also need to be defined; thus we get *D@L, D!L, C@L,* and *C!L.*

If these few @s and !s comprised the entire external address word set, then this naming scheme would not be so unattractive. A copy of the entire memory reference set, including *CMOVE, CMOVE>, MOVE, SCAN, SKIP, CSET, CTOGGLE*, etc., should be defined if the extended memory is to be used effectively.

Alternatively, only @, *!*, and the move operators might be defined as long. Then, any data at an external segment is moved to the default segment for processing. But this continual transferring is



inefficient. *L* suffix commands still might be used, but in addition to doubling the number of memory reference commands, the *L* suffix makes many of the otherwise very readable command names much less readable.

A hint of another solution to the external addressing problem might be taken from the assembler for one of the segmented processors, the 8086. The 8086 assembler uses the *FAR* prefix to indicate that an operation is to include an explicit segment reference. Using this concept we only need the named operators, each with a second method for the explicit external address reference operation. The selector suggested for this method is, as in the assembler, *FAR*. Thus *@L* becomes *FAR @*, *!L* becomes *FAR !*, etc.

The simple memory operators aren't the only candidates for *FAR*. Almost any command that takes or returns memory addresses is a candidate. Many systems have virtual memory management block buffers in external memory that are automatically transferred to local memory when accessed. This transferring is a waste of time in many cases, especially when only a small portion of the block is needed. To access the external disk buffer, the *FAR* prefix can be used on *BLOCK* or *BUFFER* to return the external address without transferring the buffer (Table 3).

New compiler options

We've seen how MCFs can be used to make code faster and shorter (*VALUE*), reduce the number of words needed in the system (*FAR, CASELESS*, and the stack operations), and make the code more readable and less cryptic (*VALUE*) as well as less complex. What else can we do with MCF words and object-oriented programming?

One hotly debated area in Forth is that of state-smart words. State-smart words test the system variable *STATE* during their execution so they may behave differently during compilation and interpretation in an attempt to behave identically in both places. While state smartness sometimes seems to simplify word usage and syntax, it also causes some hidden problems.

One of the most apparent Forth-79 state-smart words is the Forth word ' ("tick"). The ' returns a pointer to the code field of the word following it. When interpreting the sequence *' <word>*, ' executes to return a pointer to *<word>*. When compiling the same sequence, ' executes (does not compile) and a pointer to *<word>* is compiled as a literal to be returned when the procedure containing the sequence executes later.

However, it is sometimes desirable to create a procedure that uses ' to return a pointer to the word following the procedure when the procedure is later executed (not the word following the ' inside the procedure). The new procedure becomes an extension of the compiler. We must prevent ' from executing during the compilation of the new procedure. The sequence *[COMPILE] '* within the procedure solves this problem by forcing ' to be compiled (and not executed during compilation). So far, so good.

When the new procedure is interpreted, it would seem that everything should behave as advertised. But if the new procedure is used inside a procedure to execute during compilation (Listing 3), the state smartness of the embedded ' will cause ' to compile a literal pointer to the word following '. This pointer will later be returned when the newest procedure is executed rather than while we are compiling. Not what we had in mind.

Now that we understand the problem, what is the solution? One solution (used in the 1983 Forth standard) is to give the word two slightly different names, one for interpretation (') and one for compilation (['']), both without state smartness.

Another solution is to use MCFs to define two separate methods for a word, one to be performed by the interpreter and one by the compiler. Since the interpreter and compiler will select the method and *STATE* will not be tested when the word executes, the state-smartness problem does not exist. (I refer to these words as "state unsmart.") Thus the sequence *[COMPILE] '* causes the interpretable ' method to be compiled. Since the interpretable method always executes to return a pointer to a word (never compiling the pointer), the problem is eliminated. Though many words are state smart (or unsmart), only a handful of words in Forth cause these problems. Nonetheless, the MCF solution is attractive.

In Part II of this article, I will discuss extending the power of the MCF *DEFER*, simplifying metacompilation, applications of MCFs for the current ANS Forth standard, and how the syntax is implemented. ■

References

1. Bartholdi, Paul. "The *TO* Solution," *Forth Dimensions*, 1(4): 38-40, 1979.
2. Lyons, George. Letter to the editor, *Forth Dimensions*, 1(5): 56, 1979.
3. Rosen, Evan. "High Speed, Low Memory Consumption Structures," *1982 FORML Conference Proceedings*, pp. 191-196.

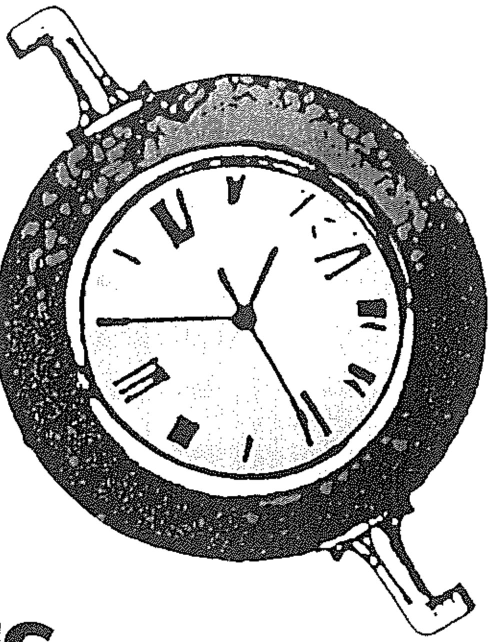
A shareware version of TaskForth, a multiuser, multitasking Forth system for the IBM PC that supports the object-oriented techniques presented in this article, is available for $25 from Shaw Laboratories, P.O. Box 3471, Hayward, Calif. 94540. (International orders add $10 for shipping and handling, Calif. residents add 6% sales tax.) The system is also available in shareware form from the COMPUTER LANGUAGE *Bulletin Board Service and CompuServe forum.*

George Shaw is a consulting software engineer and psychotherapist and has been programming in Forth for nine years. He is director of the Asilomar Forth Modification Laboratory (FORML) conference, chairman of the Silicon Valley and North Bay chapters of the Forth Interest Group, a member of the ANS Forth committee, and founder of the ACM SIG Forth (in progress). His company, Shaw Laboratories, provides microcomputer support and programming services.

Artwork: Earl Flewellen

Part 2

Forth Shifts Gears



By George W. Shaw

Object-oriented programming techniques have existed in Forth for almost a decade, though they were never recognized as such. As with the development of much of Forth, this capability has grown as a generalized solution to a specific problem.

This problem involved the transportability of pointers and a dissatisfaction with syntax in constructs like:

```
X @ Y @ +   Z !
```

Charles Moore, Forth's inventor, suggested an alternate syntax that became known as the *TO* solution:

```
X Y + TO Z
```

This syntax removed the explicit references to pointers and reduced the number of operators. Several run-time bound and compile-time bound implementations of the *TO* solution were published after its introduction, but most only addressed this specific problem.

With the generalization of the *TO* solution and greater awareness of the principles of object-oriented programming, it became apparent that the transportability and syntax problems were best solved with object-oriented solutions. Through the development of more efficient implementations of the *TO* solution, the object-oriented capabilities of Forth evolved.

Forth's object-oriented tendencies can be seen in the standard *CREATE DOES>* (or *CREATE ;CODE*) pair, which allows the definition of a single method upon the *CREATE*d object. What is needed is a syntax and mechanism to expand the *DOES>* family to allow it to define additional methods.

Normally, each structure has just one execution procedure (method) associated with it. A single code field is required to specify the behavior of the object. However, in the *TO* solution, each object *X*, *Y*, and *Z*, allows at least two methods: fetching and storing. For each method a code field is defined to specify the behavior of the method on the object. Thus *X*, *Y*, and *Z* each have at least two code fields. Objects such as these have come to be known as multiple code field (MCF) words.

MCFs give us the essence of object-oriented programming in Forth. Each code field specifies a method to be used on an instance of an object. Properly implemented, methods can be inherited. *TO* is a method selector. By using selectors rather than explicit operators, messages are passed through the compiler and interpreter, then received by the desired object. Thus data abstraction is maintained. Further, the selectors are logical operations that can be applied to a wide range of structures.

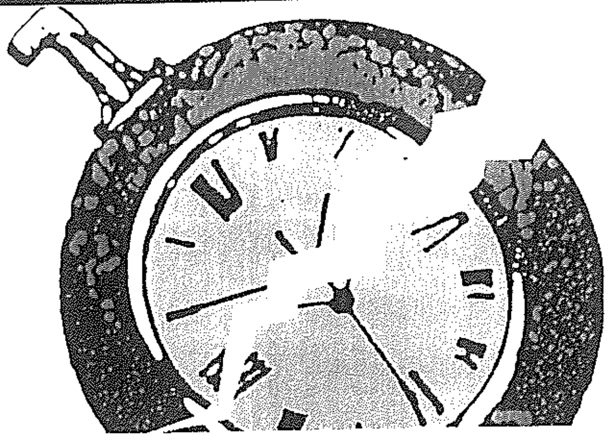
In Part 1 of this article (*COMPUTER LANGUAGE*, May 1988, pp. 67-75), I discussed many different ways to use MCFs and object-oriented programming techniques in Forth, including *VALUE*s, new stack operations, new syntax possibilities, external memory operators, *DEFER*ed words, and new compiler options. In Part 2 I will delve further into the enhancements of *DEFER*ed words and the compiler options and show how MCF words are implemented.

More powerful *DEFER*

One of the original MCF words (*VECT*) was the equivalent of a *DEFER*ed word. A *DEFER*ed word is an alterable reference to an object that is later defined or changed during run time. The simple syntax for manipulating a *DEFER*ed word is listed in Table 1 (and was discussed in Part 1). This syntax works quite well and is sufficient, as far as it goes. It is very limited, however.

The syntax previously suggested is simple but does not directly allow one to *DEFER* an object and still apply methods to it. Why? Because the selectors operate on the *DEFER* data structure, not the object *DEFER*ed. Thus additional methods of manipulating *DEFER*ed words are desirable.

If normal selectors are to perform their operations on the *DEFER*ed object, a new syntax must be defined to manipulate the *DEFER* data structure. For completeness, access to the vectored object as well as a default vectored object (for initialization and error handling) might be desired. By defining method selectors that increment the selector number rather than setting it, the selectors



VECTOR and *DEFAULT* can be used with other selectors to access the desired methods of the *DEFER*ed word (Table 1). The selector number is then placed out of range of the values possible for the *DEFER*ed object within.

Figure 1 shows the organization of the more powerful *DEFER*ed word. The code fields at positive offsets from code field zero are the methods that manipulate the *DEFER* data structure. Being positive offsets, they are not detected by error checking for out-of-range selector values (too large a negative offset) applied to the *DEFER*ed word.

The normal code fields must be coded to skip the code fields at positive offsets. This allows them to gain access to the *DEFER*ed object in the parameter field. Further, since the *DEFER*ed object can change at run time, the normal code fields must execute to bind to the proper object at run time. This allows any object to be *DEFER*ed.

Metacompiling

In Part 1 I discussed the problems that arise when a programmer wants to use the interpret time behavior of a state-smart word while it is embedded in a new word that is to be executed at compile time. In this situation, the state-smart word sees the compilation state and performs its compile-time operation rather than the desired interpret-time operation.

One solution to the problem is to make the word state-unsmart. That is, to give the word different interpret and compile time behaviors without testing *STATE*.

A state-unsmart word contains two code fields, one for interpret time and the other for compile time. When the interpreter loop finds a word, it executes the word's interpret-time code field. When the compiler loop finds a word and a compile-time code field exists, the word's compile-time code is executed; otherwise, the word is compiled. Since *STATE* is never tested, the state-smart problem doesn't occur.

This state-unsmart technique has many applications. One feature of Forth is its ability to compile another version of itself. This process, referred to as metacompilation, is rather complex to implement. The complexity comes from the normal practice of extending the compiler in Forth. If a word is defined in the child (target) system that extends the compiler, the parent (host) metacompiler must be programmed to know not only how to compile the child's structure but how to emulate its execution if the structure is used later in the child.

When metacompiling a new Forth system, this difficulty primarily occurs with : (colon), ; (semicolon), *BEGIN, WHILE, REPEAT, UNTIL, IF, ELSE, THEN, DO, LOOP, VARIABLE, CONSTANT,* and the dreaded *VOCABULARY.* (*VOCABULARY* is particularly difficult because it requires the metacompiler to compile into several different word lists, rather than just one.)

One common implementation scheme is to have all words be state-smart and either compile themselves to the child system (on disk) or execute on the parent. This can cause the same problems listed for "tick" (') (defined in Part 1) but, most annoyingly, requires a conditional test of *STATE* to select the compile-to-child or interpret-on-parent procedure in the definition of every compiler extension. This makes the words unnecessarily complex. The two-code-field, state-unsmart approach allows the interpret and compile methods to be separated and the attendant problems to be eliminated.

Solutions for the ANSI standard?

The *TO* solution was suggested as a resolution to incompatibility of pointers among systems. Because MCF words such as *STATE, BLK,* and *BASE* hide the implementation of the function and structure used, they can greatly enhance transportability in areas other than pointer usage, such as data base fields; *STATE, BLK,* and *BASE*; and other variables.

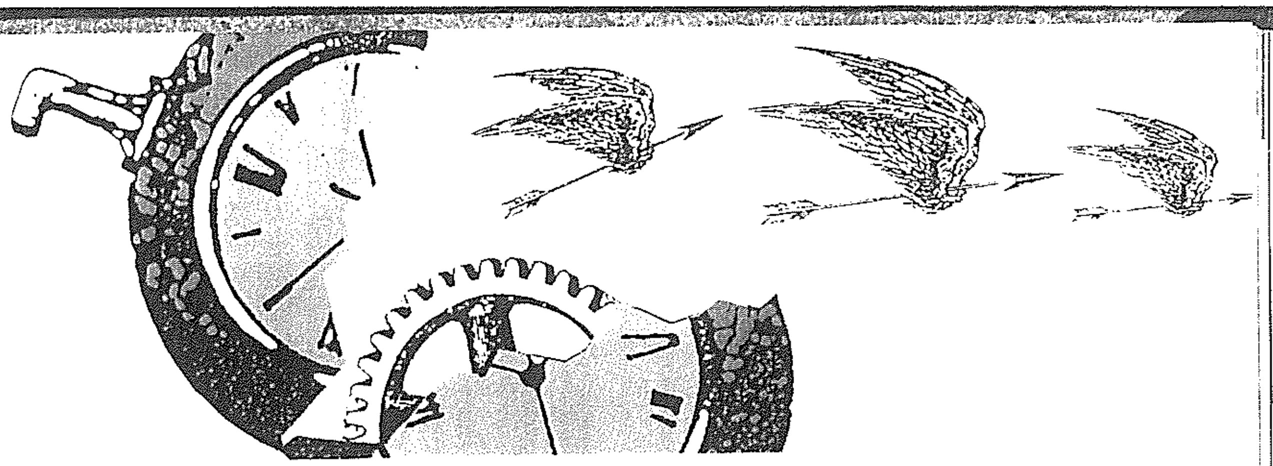
Data base fields. If data is to be exchanged among Forth systems on different processors, @ and ! cannot be used to place binary information on the storage media. The order of bytes within integers differs among processors. Data stored by a 68000 cannot be fetched by an 8086, for instance.

By defining an arbitrary byte order and using MCF data base fields, the byte order can be made consistent across systems regardless of the native @ and ! byte order. Other than byte order, the data base fields are defined just like a

MCF *DEFER*ed word syntax

Simple usage:		
DEFER <name>	(-	Defines name to be deferred)
USER DEFER <name>	(-	Defines task deferred name)
LOCAL DEFER <name>	(-	Defines locally deferred name)
<name>	(?	Executes assigned object)
TO <name>	(a -	Sets name to execute word at *a*)
Enhanced usage:		
DEFER <name>	(-	Defines name to be deferred)
USER DEFER <name>	(-	Defines task deferred name)
LOCAL DEFER <name>	(-	Defines locally deferred name)
<name>	(?	Executes assigned object)
TO <name>	(?	Performs *TO* method on object deferred)
OF <name> etc.	(?	Performs *OF* method on object deferred)
TO VECTOR <name>	(a -	Set name to execute object at *a*)
OF VECTOR <name>	(- a	Return current object)
DEFAULT <name>	(?	Execute default object)
TO DEFAULT <name>	(a -	Set default to execute object at *a*)
OF DEFAULT <name>	(- a	Return default object of name)

Table 1.



VALUE or *STRING* except the storage area is offset from a record buffer pointer.

STATE. Implementation and environmental dependencies other than byte order and pointers can be hidden and optimization performed using MCFs. For instance, standard systems cannot change the value of the variable *STATE* to control compilation and interpretation. *STATE* can only be an indicator for the system. Several vendors have even implemented the compiler in ways that make properly maintaining *STATE* a nuisance.

If standard compiler and interpreter control is to be allowed, additional words, techniques, or rules need to be defined. This is not desirable. One solution is to make *STATE* an MCF word. The sequences:

```
TRUE TO STATE    ( start compiling)
FALSE TO STATE   ( start interpreting)
```

can do whatever is necessary to put the system in the selected state. For efficiency and consistency with other MCF words, *STATE* should then be defined to return its value rather than a pointer to its value. However, for compatibility with the previous standards, a pointer can be returned with an additional method to return the value, if desired.

BLK. The variable *BLK,* which indicates which mass storage block is being interpreted or compiled, is another candidate for definition as an MCF word.

According to the standard, when *BLK* is zero, input is coming from *TIB*, the text input buffer (usually the terminal). When *BLK* is not zero, input is coming from the mass storage block. This behavior requires a run-time conditional test to direct the input stream to the correct place. If *BLK* is made an MCF word, then the sequences:

```
0 TO BLK     ( set terminal input stream)
100 TO BLK   ( set mass storage input)
             ( stream to block 100)
```

can allow the system to fix the input stream at the *TIB* when *BLK* is set to zero and always cause a block lookup when *BLK* is not zero. This would speed interpreting and compiling, especially if the interpreter and compiler are imple-

Enhanced MCF *DEFER*ed word memory layout

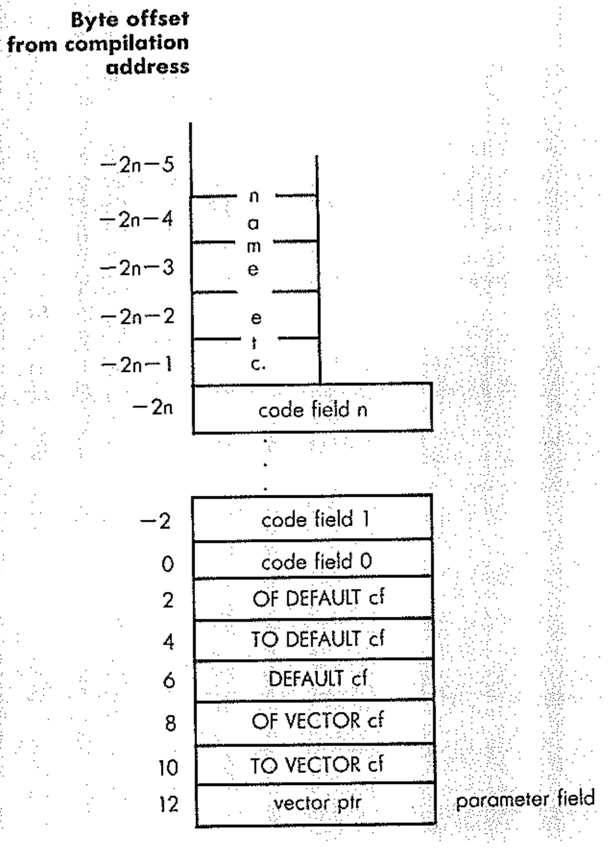


Figure 1.

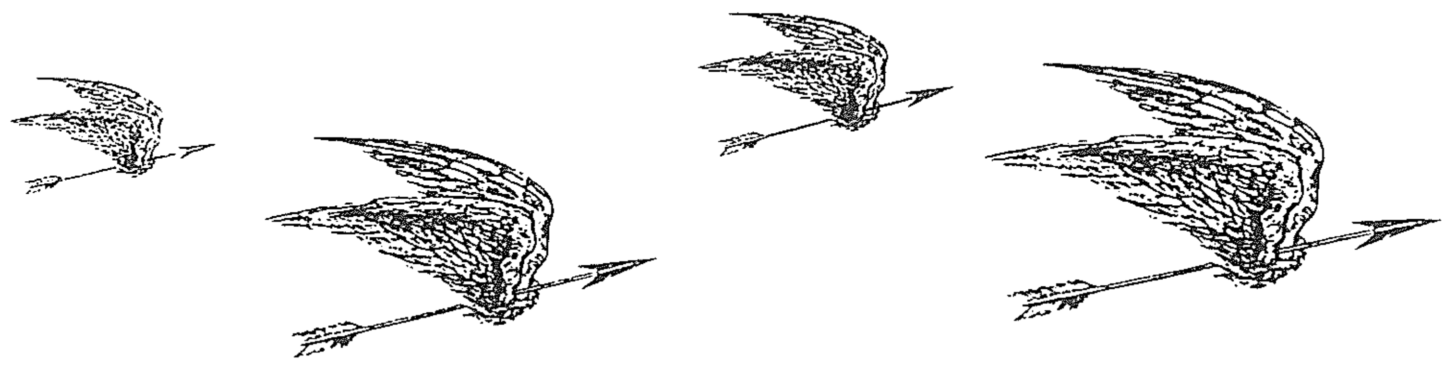
MCF operational behavior

Compilation:
Immediate words are executed during compilation to control the compilation process. To allow greatest control, the method selector is not applied to the immediate word before execution. The immediate word may then use or ignore the method selector, as needed.

Interpretation:
For consistency with compilation, the method selector is not applied to immediate words before they are executed.

Compilation enhancement behavior
During compilation, if an immediate word has exactly two methods (code fields), the second method is executed rather than the first. Or if a word is immediate and its enhancement bit is set, the second method is executed rather than the first.

Table 2.



mented with the conditional test in high-level Forth rather than assembler.

BASE. *BASE* can be treated just like *BLK*. The variable *BASE* specifies the radix for number input and output. Forth allows any number base to be used. With *BASE* as an MCF word, the system can be optimized for converting in the given base. Decimal, hexadecimal, octal, and binary are the most likely candidates to optimize; others can be performed with the current general conversion routine.

Other variables. In Forth, values are fetched much more often that they are stored. To deal with this, MCF words were developed (also discussed more thoroughly in Part 1). The fetch/store frequency holds true for all variables listed in the Forth standards (including *SPAN* and *#TIB*), and any others that might be added by the current ANSI Forth standards committee.

If the standards committee is interested in eliminating more words from its document, the ." and .(pairs, as well as ' and ['], could be consolidated into one word: ." and ', respectively. This could be done by defining them as state-unsmart words.

Other questions surround the proposed standard. What of the pointer problems and external address operators described in Part 1? Are these to be available in the ANSI standard? Or are 16-bit systems to be crippled without a standard mechanism for external addressing? Are we to add a large set of *L* operators that 32-bit systems would be required to include, even though 32-bit applications would never use them?

With *FAR* operators, 32-bit systems would not require additional names and could simply manipulate the extra address bits to emulate a segmented 16-bit data address space. Large, 16-bit applications can be ported to 32-bit systems. 32-bit applications wouldn't care because they wouldn't use the *FAR* operations. The *FAR* concept seems more attractive and is, in fact, the syntax used by many C compilers to solve the same problem.

How does it work?

As you have seen, the defining syntax for the MCF words described is upwardly compatible with the existing Forth standards and systems. The syntax works by

MCF word memory layout

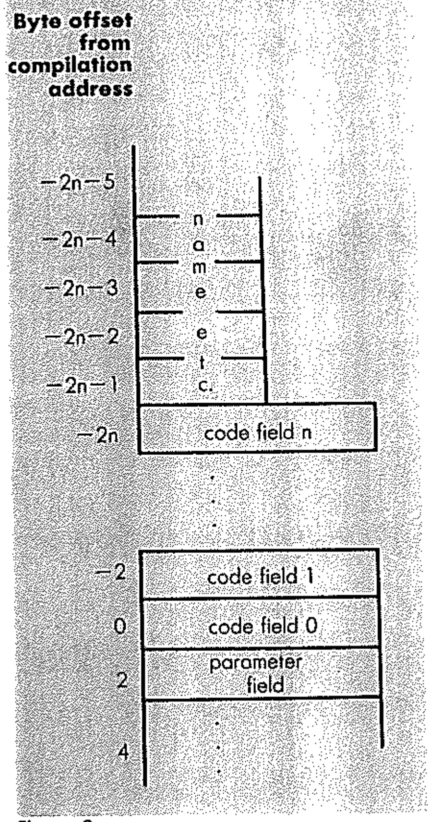
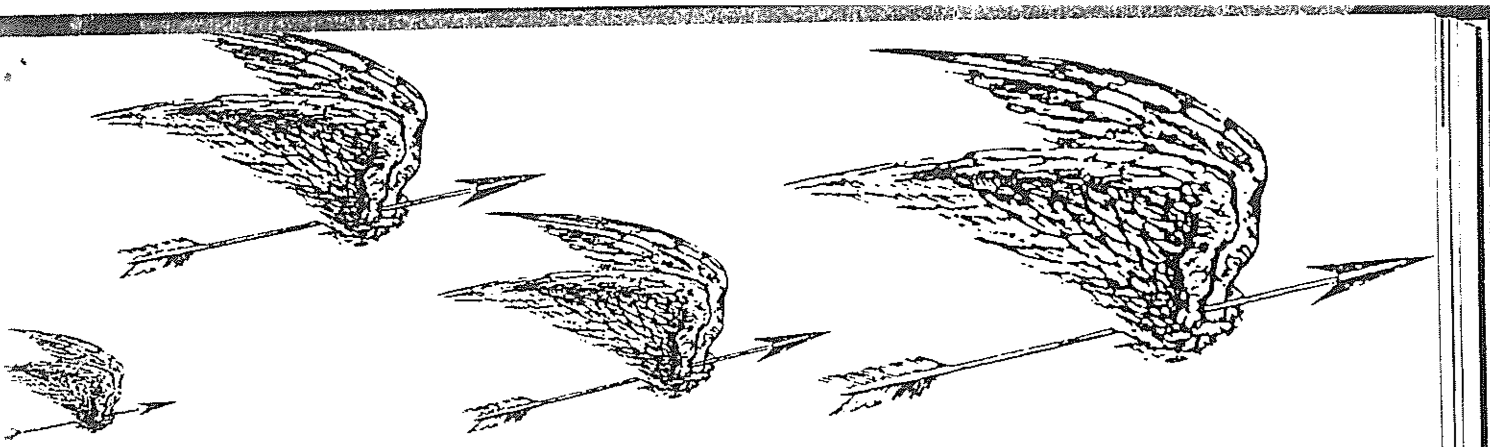


Figure 2.

```
HEADER A-VALUE      ( one-time VALUE using interpretable DOES) )
   3 CODEFIELDS  0 , ( data place holder)
   OF DOES)   ;
   TO DOES)   !   ;
   DOES)   @   ;

HEADER SP           ( special case stack pointer access )
   2 CODEFIELDS
   TO ;CODE   SP POP   NEXT   END-CODE
   ;CODE   SP AX MOV   APUSH   END-CODE

HEADER STATE        ( transportable compiler control )
   2 CODEFIELDS   FALSE , ( flag storage )
   TO DOES)   OVER SWAP !   IF ( set compiling) ELSE ( set interpreting)
   THEN ;
   DOES)   @   ;

State-unsmart examples of previously state-smart words
: :)         ( define high level run-time )
   [COMPILE] DOES)   COMPILE DROP   ;   IMMEDIATE
ALIAS COMPILER TO  ( more readable)

HEADER ."   ( - )( display following text )
   2 CODEFIELDS   IMMEDIATE
   :)   "TEXT TYPE   ;     ( "TEXT parses a " delimited string )
   COMPILER :)   COMPILE (.")   "TEXT   PLACE,   ;

HEADER '   ( - a )( obtain address of word following )
   2 CODEFIELDS   IMMEDIATE
   :)   BL WORD   FIND   0= IF ( not found error ) THEN   ;
   COMPILER :)   RECURSE ( execute normal self) [COMPILE] LITERAL   ;
```

Listing 1.



following the behavioral conventions listed in Table 2.

The new execution procedures are defined with *DOES>* and *;CODE*, just as they are now. The only difference is that the method selector becomes a prefix for *DOES>* and *;CODE* to tell these words to compile code that will build the appropriate code field when the defining word is executed. *CREATE* must be internally split into *HEADER* and *CODEFIELDS* to allow a variable number of code fields to be defined.

DOES> and *;CODE* should also be made state-smart (or state-unsmart with MCFs) to allow run times to be defined interpretively, but this is not required. Interpretively defined run times do not leave behind a compiled creation-time portion or headers of the run times and defining word, thus saving additional memory. The structure of a complete MCF dictionary entry is given in Figure 2.

The interpreter and compiler as well as the systems dictionary search routine (*FIND*) need to be modified slightly. *FIND* must return a pointer to code field zero. Thus *FIND* must skip the other code fields before returning the pointer. The variable *CF-* is added to the system to hold the current method selector value for use by both the interpreter and compiler.

If a found word is not *IMMEDIATE* (designated to execute during compilation instead of being directly compiled), the method selector value is subtracted from the CF0 address and the selector set to zero. This new CFx address is executed if the program is interpreting or compiled if it is compiling. The method selection operation is very fast and simple so the effect on compilation speed is negligible.

Selectors such as *TO* and *OF* simply set the method selector variable *CF-*. Thus *CF-* is actually an MCF object (procedure and variable). It is set to the negative offset needed to reach the desired code field. *CF-* is defined to be set by *TO*. Otherwise it executes by subtracting its value from the given CF0, returning the result and then setting itself to zero. If error checking is desired to ensure that the code field exists, it can be performed in *CF-*. Compiled and interpreted defining examples are given in Listing 1.

A few minor points: *OF* (originally called *AT*) is somewhat unnecessary with *VALUE*s. When a pointer to a *VALUE* is desired, try to use a *VARIABLE* instead. The temptation to use *+!* with a *VALUE* pointer is too great. One can not be sure that the pointer is valid for *+!*. The *+TO* approach for incrementing the value is safer because it maintains the data abstraction by keeping the inside of the *VALUE* object hidden. When a pointer is needed, use ' and run-time binding. When run-time binding to an object is desired, ' the object, perform the method selection directly, then *EXECUTE* the resulting code field pointer. For example:

```
: FOO
   ' ( address word following FOO)
   [COMPILE] TO
      ( obtain TO selector)
   CF-  ( perform selection)
   EXECUTE      ;
```

Applying the techniques

MCF words allow an appropriate mechanism for using prepositions in Forth syntax; in fact, most of the defined selectors have been prepositions. C and other languages can hardly approach Forth's ability to extend, modify, and simplify itself as techniques and styles advance.

I have used these techniques in applications for almost four years and have found them to be of great help. For example, in an inventory application with over 120 fields (objects) in 20 files, these techniques allow the creation of a generic set of input verification routines for each format of object (15 or 20) rather than for each individual object. The input form procedures execute the fetch operation on the object and convert it to ASCII, allow editing, then verify the value and execute the store operation on the object.

Those who use objects and share method selectors heavily may find it useful to translate the method selectors from a logical selector number to a physical code field offset for each structure class. This can easily be performed by storing a pointer to a translation table between the header and the code fields, thus reducing side effects and enhancing the reusability of the selectors.

These techniques have their advantages and disadvantages (Table 3) and offer great promise for transportability and implementation independence. The abstraction allows a vendor to implement the internal data structures of his or her system to gain any level of efficiency

Pros and cons of MCF words

Advantages	Disadvantages
Less memory needed when used	Slows compilation slightly
Less memory needed to define (sometimes)	System memory overhead might not be recovered
Executes faster (usually)	More memory needed to define (sometimes)
More readable	New concept/syntax to learn
Class specific operators not needed	More syntax options to consider
Class spanning operators not needed or are simpler	Not widely supported
Generic operators definable	Until widely available, reduced code transportability
Code complexity reduced	As a system extension, code may be slower
Syntax options expanded	Some new words required
Enhances transportability of code	*TO* is not Forth-like
Enhances transportability of data	Shows other language influence
Greater regularity in usage	
More compact source code when used	
Methods centralized and bundled	
Implementation independence/versatility	
Function optimization possible	

Table 3.

Eco-C88 C Compiler with Cmore Debugger

Professionals prefer the Eco-C88 C compiler for ease of use and its powerful debugging features. Our "picky flag" gives you nine levels of lint-like error checking and makes debugging easy:

"I'm very impressed with the compiler, editor, and debugger. I've tried quite a few different compilers for the PC and have given up on all of the others in favor of yours... I've gotten to the point where I download C code from a DEC VAX/VMS system just to be able to compile it with the picky flag set at 9. It finds lots of things VMS totally ignores..."
JS, Oak Ridge, TN

The Eco-C88 compiler includes:

- A full-featured C compiler with 4 memory models (up to 1 meg of code and data) plus most ANSI enhancements.
- Without a doubt, the best error checking you can get. We catch bugs the others miss, making you much more productive.
- Cmore is a full-featured source code debugger, not some stripped-down version.
- Robust standard library with over 230 useful (no "fluff") functions, many of which are System V and ANSI compatible. Full source is available for only $25.00 at time of order.
- CED, a fast, full screen, multiple-window program editor with on-line function help. You can compile, edit, and link from within CED.
- cc and mini-make utilities included that simplifies the most complex compiles.
- Users manual with over 150 program examples (not fragments) to illustrate how to use the library functions.
- Fast compiles producing fast code.

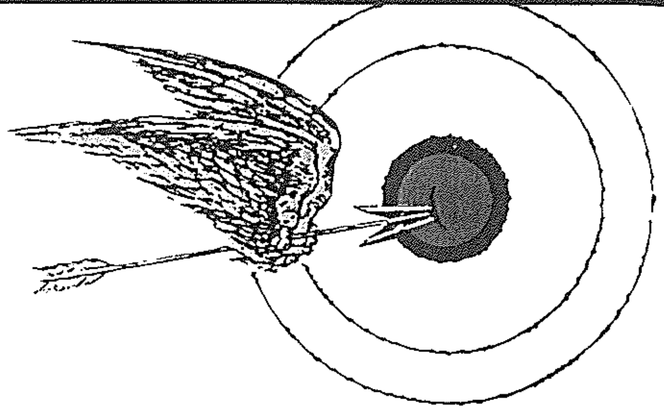
Our Guarantee: Try the Eco-C88 compiler for $99.95. Use it for 30 days and if you are not completely satisfied, simply return it for a full refund. We are confident that once you've tried Eco-C88, you'll never use anything else. Call or write today!

Orders: **1-800-952-0472**
Info: **1-317-255-6476**

Ecosoft Inc.
6413 N. College Avenue
Indianapolis, IN 46220

ECOSOFT

CIRCLE 29 ON READER SERVICE CARD



available yet still remain transportable. Segmented memory spaces or external memory spaces can be used and their effect hidden from the application. This transportability technique may be one of the few that increases efficiency. ■

George Shaw is a consulting software engineer and psychotherapist and has been programming in Forth for nine years. He is director of the Asilomar Forth Modification Laboratory (FORML) conference, chairman of the Silicon Valley and North Bay chapters of the Forth Interest Group, a member of the ANS Forth committee, and founder of the ACM SIG Forth (in progress). His company, Shaw Laboratories, provides microcomputer support and programming services.

Artwork: Earl Flewellen

Reading punctuation in Forth

The implications of punctuation or attached prefixes and suffixes in Forth identifiers are by convention only. The compiler is not affected by the punctuation or attached prefix/suffix style. All punctuation is allowed.

'	Pronounced "tick," this operator returns a pointer to the word following it. As prefix of an identifier, ' (tick) implies "pointer to"; as a suffix of an identifier, ' (prime) implies "difference from."
,	The comma operator compiles 16-bit values into the dictionary and moves the dictionary pointer forward. Prefix characters indicate the operator is modified for size (for example, *C,* for characters). As part of an identifier, it implies dictionary extension.
@	This operator reads 16-bit values from memory given a pointer. Prefix characters indicate the operator is modified for size (for example, *C@* for characters). As part of an identifier, implies a device, memory, or port read.
!	! stores 16-bit values into memory given a pointer. Prefix characters indicate the operator is modified for size (for example, *C!* for characters). As part of an identifier, implies a device, memory, or port write.
+!	This operator adds 16-bit values to memory given a pointer. Prefix characters modify the operator for size (for example, *C+!* for characters).
()	This operator is used to enclose comments (this is a comment). As a prefix or prefix/suffix, it indicates the runtime or internal identifier name, not normally used directly.
"	This operator compiles strings in-line or delimits the end of a string. As a prefix of an identifier, it implies string or string operation; as a suffix of an identifier, it implies that a quote-delimited string is expected.
[ ]	Operators to temporarily leave and reenter compilation to calculate a value or other interpretable task. As a prefix/suffix pair (such as *[']*), it implies an operation to be performed during compilation.
;	This operator ends a high-level definition. As a prefix or suffix, it implies a compilation structure ending.
:	This operator begins a high-level definition. As a prefix or suffix, it implies compilation structure beginning.