Multiple Code Field Data Types and Prefix Operators

Klaus Schleisiek

POB 202 264
2000 Hamburg 20
West Germany

Abstract

This paper presents compilers for creating "intelligent" data types which have more than just one code field, as words in classical FORTH do. These compilers introduce a prefix syntax into FORTH to create a whole class of data structures, the first of which became known in the FORTH community as the 'TO' concept in 1979.

Prefix Syntax in FORTH

FORTH is an RPN language with a few exceptions. Whenever a string of characters is expected from the input stream, a prefix operator is used, e.g. ' : or ." . Another case is the word COMPILE which precedes the word which is supposed to be compiled. These are cases where the prefix syntax is either inherent in the nature of the problem (expecting a string from the input stream) or leads to an elegant implementation of a function (COMPILE <name>).

In 1979, the first paper appeared [BAR79] proposing a prefix operator for a data type, namely TO. TO was supposed to change the behavior of a VALUE, which usually would behave like a CONSTANT, but when preceded by TO, it would act equivalently to <variablename> ! . The reason for introducing this structure was twofold:

a) to get away from manipulating addresses, hence making programs more portable,
b) to enhance the readability of programs.
(A TO B versus A @ B !)

This concept did not gain acceptance due to its problems:

a) The way it was implemented resulted in slower execution compared to using VARIABLEs, especially in a multitasking environment.
b) There was no simple means of getting the address of the TO variable, which, amongst other things, gave rise to the prefix +TO.
(1 +TO A versus 1 A +!)

Shortly thereafter, a proposal was publised [LYO80] to switch the decision process of how a prefixable data type ought to behave from execution to compilation time by using data types with more than one code field and compiling the appropriate code field, depending on the prefix. This proposal was not actually implemented until 1982 when it was shown [ROS82] that it has its merits.

Rosen presented two data types, QUAN and VECT, each having three code fields acting as follows:

```
QUAN A                  VARIABLE A
  A          ==           A @
TO A         ==           A !
AT A         ==           A

VECT A                  VARIABLE A
  A          ==           A @ EXECUTE
TO A         ==           A !
AT A         ==           A @
```

This technique results in expressions which not only compile into fewer bytes, but also execute faster (if implemented in machine code) than the equivalent phrases using VARIABLEs. Although multiple code field words (MCF-words) with their prefixes do not add any new features to FORTH, their alternative syntax can enhance program readability. They also execute faster and take less space than their single code field counterparts. This gave rise to developing the following tools which enable a programmer to utilize prefixes and MCF-words in a readable and transportable way:

a) A defining word for creating prefixes which will select a certain code field of their associated MCF-word, including an error check on data type compatibility.
b) A set of words for defining defining words that create MCF-words, associating a certain prefix with each code field.
c) A word which gains access to the parameter field address of an MCF-word from its compilation address.
d) A word which allows one to tick a certain code field of an MCF-word.

The following discussion is based on an implementation that exhibits the following characteristics:

- 8-bit processor (hence a preference for accessing bytes)
- Indirect threaded code
- Preincrementing address (inner) interpreter
- FORTH-83 Standard, notably ' returns the compilation address of a word (formerly called "code field address").

Other implementations (DTC, ITTC, JSR-threading) have not been evaluated as yet. Multiple code fields will likely be more costly in terms of memory usage in DTC and JSR systems though.

Syntax of Proposed Words

a) Prefixes

A prefix is a state-smart, immediate word which is only valid if used immediately preceding its associated MCF-word. Hence an error check is included in every prefix, testing its applicability to the word following in the input stream. A prefix selects a certain code field of the word it precedes, and, depending on the state of the system (compiling or interpreting), either compiles or executes it. Prefixes are created using the following syntax:

```
+n Prefix <name>
```

where +n is the code field number identifying the code field to be selected. Code fields are counted starting from zero. Note that a prefix which was defined with +n equal to zero will select the first

code field of the MCF-word following, which has the same effect as using the MCF-word without a prefix.

b) Defining Multiple Code Field Words (MCF-words)

A defining word has to be defined which exhibits several DOES> parts, so to speak, one for every code field of the MCF-word to be created by this defining word. Picking up on a proposal for a new syntax for defining defining words [RAG80] paved the way for a solution. The behavior of the code fields of an MCF-word is defined prior to the definition of the constructing part of the data type (also called the "create part"). These code field routines are coded using DO> or CODE>, depending on whether they are defined in high level or machine code. The constructing part of the data type is a colon definition including the immediate word BUILD, which will compile information left by DO> and / or CODE>, and later on will create a dictionary entry and compile the specified code fields.

Hence the following syntax emerged:

```
n codefields
    ' <prefix.n-1> Do>     ... ;
    ' <prefix.n-2> Do>     ... ;        n code field
        .          .       ... .        definitions
    ' <prefix.1> Code>     ...  end-code
                 Do>       ... ;

: <name>  ...  build  ...  ;
```

Later on using the phase:

```
<name> <namex>
```

will create the data type <namex> with n code fields.

Preceding DO> or CODE> with the compilation address of a prefix results in the association of the prefix with this codefield routine, which makes possible the error check mentioned earlier. Similar to DOES> in classical FORTH, DO> lets <namex> push its parameter field address on the stack at execution time. But there is no longer a fixed offset between one of the several compilation addresses of <namex> and its parameter field. Instead it has to be computed, and the offset depends on the location of the code field being executed within the series of code fields of an MCF-word. This offset is compiled by DO> or CODE>, given the information on how many code fields there are altogether (n) and the rank of DO> or CODE> within the sequence of code field definitions. Note that the outermost code field is defined first and the first code field last. Every definition of a code field leaves its code address on the stack, which later on will be compiled appropriately by the immediate word BUILD within the definition of <name>.

c) Locating the Parameter Field Address

The word >BODY in FORTH-83 computes the parameter field address, given a word's compilation address. This is no longer trivial if words may have a varying number of code fields.

The solution is two-tiered:

1. Another word, >MBODY, is defined which returns the parameter field address of MCF-words. This is simple given the information compiled by DO> or CODE>. But the programmer has to take care of using the "proper" >BODY for a data type.
2. The system has to be recompiled such that every relevant code field routine is preceded by its parameter field offset, according to the structure compiled by DO> or CODE> such that, for example, ' <constantname> >BODY returns the same as ' <constantname> >MBODY outlined under 1. Then >BODY may be implemented the "smart" way.

Note that the Standard does not specify >BODY such that it may be used to locate the beginning of machine code from the compilation address of a word implemented in machine code in ITC systems. However, it may work in many ITC systems and be misused to that end.

d) Ticking (') the rth Code Field of an MCF-word

The word ' 'gives access to code fields other than the first one of an MCF-word <namex>. The phrase:

' ' <prefix> <namex>

returns a compilation address of <namex>, the one which is associated with <prefix>. Unfortunately a new word has to be introduced. The alternative, to make 'smart enough to recognize prefixes and act accordingly, would no longer allow one to tick a prefix itself.

Implementation

Implementing MCF-words in a transportable way requires one to write a piece of machine code to make DO> work properly. When a code field gets executed which was defined in terms of DO> , a small piece of machine code precedes the threaded addresses of the high level definition for that code field. Generally, this would be a JSR-instruction to a routine called (DODO , though in many ITC systems a JMP (DODO will suffice (this is true for Z-80, 8080, 8086). A snapshot of the state of the FORTH pseudo registers I and W after the execution of NEXT, but before the execution of the second code field of the MCF-word <namex>, will reveal the kind of function to be performed by (DODO (Fig.1).

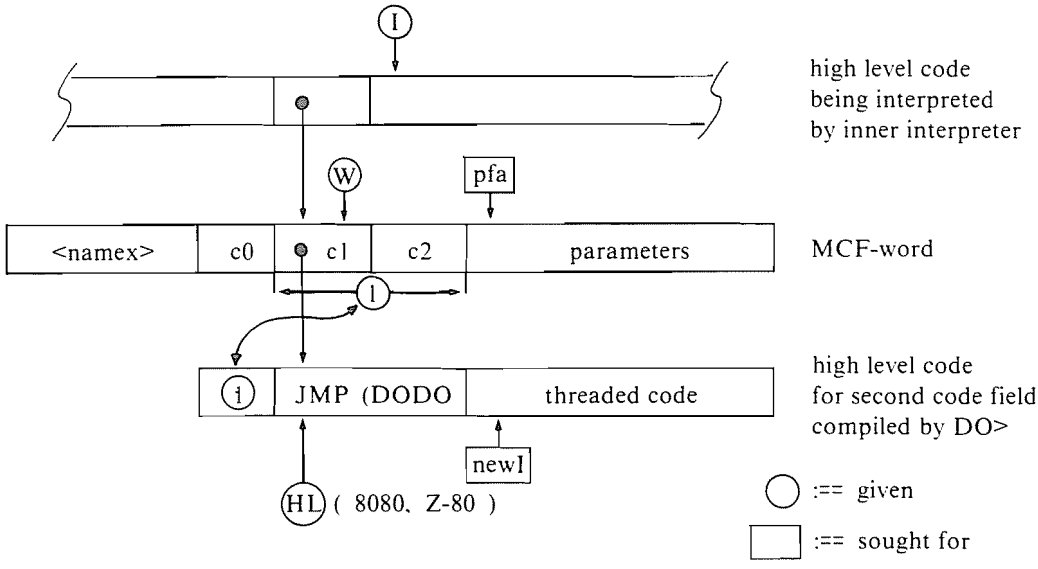

Fig. 1

This discussion is based on a Z-80 processor. The pfa has to be pushed on the stack pointing to the parameter field of the data type <namex>. This address can be computed by adding 1, which had been compiled by DO>, to W. It has to be pushed on the return stack, and newI is the content of HL incremented by three.

Compiling Defining Words that Create MCF-words

One primitive word, (CODEFIELD, handles the compiling part for both DO> and CODE>. Its input parameters are, from bottom to top, #code, the number of code fields to be generated altogether, and #sequence, the code field number of the latest code field routine defined, and

optionally the compilation address of a prefix. Note that prior to compiling the first code field routine, #sequence is initialized to equal #code; the word CODEFIELDS simply does a DUP. The presence of a prefix address is detectable since #sequence will be a "small" number (at most eight in the current implementation, allowing for at most eight code fields) which is well below any compilation address of a prefix. Note that this may not automatically be true in an ITTC system. If a prefix address is present, its parameter field address is compiled into the dictionary to allow for error checking on data type compatibility (see below). Then #sequence is decremented by one, and now the current state of #code and #sequence allows one to compute the byte offset of the code field being defined to the parameter field address of the MCF-words characterized by this code field routine. This byte offset is compiled into the dictionary. Now HERE points at what will become the code address of the current code field routine, and it is rotated under #code and the updated #sequence to be passed to BUILD, within the constructing part of the current data type definition. Then in the case of DO>, JMP (DODO is compiled, and the system enters compile state; in the case of CODE>, the Assembler becomes the first vocabulary in the search order.

BUILD compiles into the colon definition of the constructing part of the data type a word which, when executed, will create a header in the dictionary without any code field. Then the word (," is compiled, followed by the code addresses left by DO> or CODE>. When (," is executed, it will compile these addresses as the code fields of the data type. Note that a simple error check on completeness of the code routine definitions is possible since #sequence will be decremented to zero upon completion. The reason for making BUILD an immediate word to be used within a colon definition (as opposed to BUILD> in [RAG80]) was to allow for computations to take place prior to creating a header in the dictionary for the new data type <name>.

Prefixes

Prefixes which were created by the defining word PREFIX tick the word following the input stream, returning its compilation address. This address happens to be the address of the first code field. Then the code field number, the parameter used to define this very prefix, is added, yielding the compilation address to be executed or compiled.

The error check mentioned above works in the following manner. If a code field routine defined in terms of DO> or CODE> was associated with a prefix, its parameter field address has been compiled into the dictionary. Hence a comparison against the prefix's parameter field address may take place, resulting in an abort sequence if there is no match. See Figure 2 for the code being compiled.

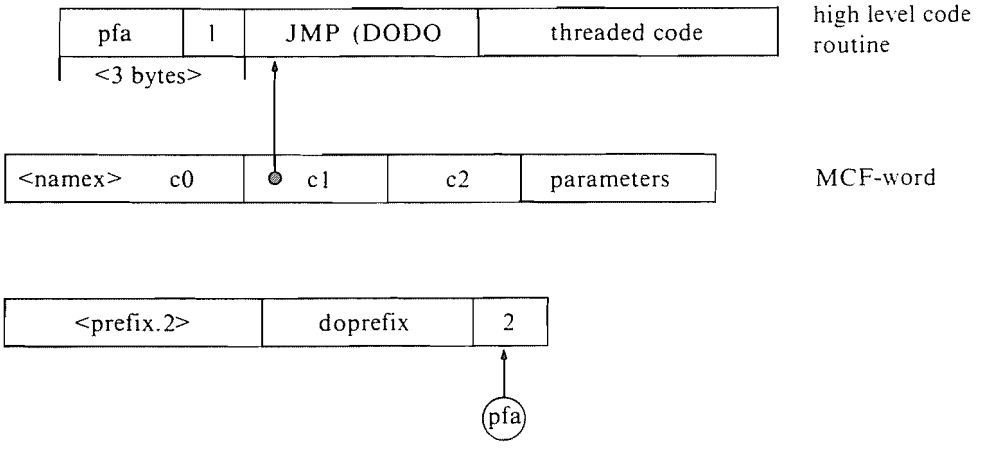

Fig. 2

Accessing the Parameter Field Address

The implementation of >MBODY is straightforward. Since the byte offset of the parameter field address, relative to a certain code routine, is compiled in front of that code routine, it can be fetched and added to one of the compilation addresses of the MCF-word.

Ticking Prefix Specific Code Fields

This is straightforward, too. Since a prefix contains the byte offset of a certain code field, relative to the first code field of an MCF-word in its parameter field, the prefix is ticked first, its offset extracted, and then the MCF-word is ticked and the offset added. Also, an error check on data type compatibility may be included, along the lines outlined under Prefix.

Conclusion

Prefix operators are usually regarded as running contrary to the FORTH philosophy. But even classical FORTH can not live without prefix syntax as outlined in the introduction. Hence I felt justified to invest the effort of creating compilers to create "intelligent," multiple code field data types and prefixes as selection operators. These may be useful in several respects:

a) To build non-FORTH-like languages on top of FORTH.
b) To utilize their inherent code compactness and execution speed if used with care.
c) To give a choice of syntax if it helps to make programs more readable.

I have the feeling that these compilers have matured far enough to be both understandable and safe to use. Due to the implementation, extensive error checking can be performed if desired. Furthermore, these structures are easily metacompiled, a feature which is inherited from the structure first outlined in [RAG80].

Acknowledgments

I'd like to thank the referees for their comments and criticism on the first draft of this paper. It resulted in a thorough revision in terms of presentation and implementation.

References

[BAR79] P. Bartholdi, "The 'TO' solution, and 'TO' solution continued," *FORTH Dimensions*, Vol. 1, No. 4/5, 1979.

[LYO80] G.B. Lyons, "Note on the 'TO' solution," *FORTH Dimensions*, Vol. 1, No. 5, 1980, p. 56.

[ROS82] E. Rosen, "High Speed, Low Memory Consumption Structures," *Proceedings FORML Conference 1982.*

[RAG80] W. Ragsdale, "A New Syntax for Defining Defining Words," *Proceedings FORML Conference 1980*, p. 122.

Manuscript received May 1983.

Klaus Schleisiek attended the Universitaet Dortmund, Germany, and studied computer science. In 1979-81, he developed a multi-channel sound synthesizer in New York, New York, getting experience with FORTH for the first time. He is currently a partner and the system designer for System Partner GmbH, Germany, a firm dedicated to developing fault tolerant turn-key business micro systems, using FORTH exclusively for both the operating system and application programs.

Appendix I

Implementation of multiple code field data types and prefix operators for Z80 and 8086.

```
Create (dodo Assembler ( Z-80 )
  X 2dec     X) A 1d     X 2inc     X 2inc     X 2inc
  W X exx     X- add      A X- mov     CS ?[  X+ inc  ]?
  X 2dec     X push     ' : @ jmp  ( exit via docolon )
end-code
          ( Note: X :== HL . W :== DE )

: dodo.     0C3 c.     (dodo . :

Create (dodo Assembler ( 8086 )
  R dec     R dec     I R ) mov     W ) I mov
  -1 I D)     A- mov     cbw     W A add     A push
  I inc     I inc     1 inc     Next     end-code

: dodo.     0E9 c. (dodo here 2+ - . :

: Prefix  ( codefield - )
  Create 2* c. immediate
  does>  '  over c@ +              ( determine code field )
         swap  over @ 3 - @
         - abort" invalid Prefix"
         State @ IF . exit THEN execute ;

: codefields  dup ;
: (codefield  ( #code #seq1 [prefix] - addr #code #seq2 )
  dup 8 u> IF >body . THEN     ( compile pfa of prefix )
  1- 2dup - 2* c.              ( compute and compile offset )
  here -rot ;                  ( pass code address )

( An alternative for error check afficionados: )
: (codefield  ( #code #seq1 [prefix] - addr #code #seq2 )
  dup 8 u> IF >body dup >r  c@  over 1- 2*
           - abort" wrong prefix"  r> . THEN
  1-  2dup - 2* c.     here -rot ;

: Do>  ( #code #seq1 [prefix] - addr #code #seq2 )
  (codefield dodo. ] ( smudge ) ;

: Code> ( #code #seq1 [prefix] - addr #code #seq2 )
  (codefield Assembler :

: header Create -2 allot ;

: (." r> count  2dup + >r            ( access in line string )
          here swap dup allot cmove ;     ( and compile it )

: build  ( addrn-1 . . addr0 #code #seq - )
  abort" incomplete"
  compile header  compile (."
  dup 2* c.  0 DO . LOOP ; immediate
```

```
: >mbody ( cfa - pfa )  dup @ 1- c@ + ;

: ' '    ( - cfa )    ' >body c@ ' + ;
```

Appendix II

Some examples of usage:

a) The classical ones

```
1 Prefix TO                2 Prefix +TO

3 codefields ' +TO  Do> +! ;
             ' TO   Do> ! ;
                    Do> @ ;
: VALUE     build 0 , ;

1 Prefix TO                2 Prefix AT

3 codefields  ' AT  Do> ;
              ' TO  Do> ! ;
                    Do> @ ;
: QUAN      build 0 , ;

3 codefields  ' AT  Do> @ ;
              ' TO  Do> ! ;
                    Do> @ execute ;
: VECT      build ['] noop , ;
```

b) Some which proved useful

```
1 Prefix ->     ( "into" )     2 Prefix [ ]     ( "content-of" )

3 codefields  ' [ ]  Do> @ ;
              ' ->   Do> ! ;
                     Do> ;
: Variable     build 0 , ;
```

This form of a Variable is "upward compatible" to the Standard Variable but allows the equivalent of @ and ! via prefixes in a more memory efficient way and with faster execution speed if implemented in code.

```
1 Prefix ->                2 Prefix make

Create restore ] r> r> ! ;

3 codefields
   ' make Do>  r> over dup >r @ >r restore >r
               dup 2+ >r @ swap ! ;
     ' -> Do>  ! ;
          Do>  @ execute ;
: Executor build ['] noop , ;
```

If used without prefix it just behaves like a VECT. Using the phrase " ' <name> -> <executor> " will assign <name> to <executor>. The phrase " make <executor> <name> " , which is compile only, will temporarily assign <name> to it until EXIT is executed which will restore <executor> to what it was before.

An example of usage:

```
Executor ?

: lost  ." press <cr>" ;
: fine  ." ok" ;            ' fine -> ?

: game make ? lost
  BEGIN key dup 0D = IF drop exit THEN
            Ascii ? = IF ? THEN
  AGAIN ;
```

c) Finally, an example which shows an unusual yet meaningful use of multiple code field words.

```
2 codefields    Do> @ >r ;
                Do> dup 2- , 2+ >r ;
: Compiler      build here 0 , ] immediate ;

: Executes compile exit here swap ! ; immediate

: ."            Ascii " word c@ 1+ allot ;

Compiler ." ." Executes r> count 2dup + >r type ;

Compiler "  ." Executes r> count 2dup + >r ;

Compiler abort" ."
Executes        IF sp! here count type space
                  r> count type quit THEN
                r> count + >r ;
```