P R E L U D E a n d F I N A L E

Implicit context switching based on pre- and post-executed words

Manfred Mahlow
Weißenburger Str. 31 , D-28211 Bremen , Germany
Phone: ++49 421 69458 65 and ++49 421 447395
email: c/o anwind@compuserve.com

Abstract

On the annual conference of the German Forth Association in 1997, I presented the "Prelude-Concept", a simple but powerfull approach to early-bind methods to data. It´s simple enough to be applied to small microcontroller systems and powerfull enough to be useful for fat systems too.

Since that time, the concept was used to implement extensions for context-oriented or object-oriented programming for several forth systems and based on that experience a more general concept of pre- and post-executed words evolved, that will be presented in this paper.

What are pre- and post-executed words ?

A pre- or post-executed word is a Forth word, that is assigned to another Forth word, to extend it´s compile-time and execute-time semantics. A pre- or post-executed word is hidden behind the word, it´s assigned to, while the word, it´s assigned to, is visible to the programmer like any other forth word. A pre- or post-executed word is executed, when the visible word is compiled or executed by the outer interpreter.

Two words are needed, to handle pre- and post-executed words, **prelude** and **finale** .

´ name prelude

assigns the word name as a pre-executed word (prelude) to the next created word and

´ name finale

assigns the word name as a post-executed word (finale) to the next created one.

Let´s have a look on Figure 1, to see how it works :

```
: word1 ( -- ) ." word1 executed " ; <enter> OK
´ word1 prelude <enter> OK
: word2 ( -- ) ." word2 executed " ; <enter> OK

word2 <enter> word1 executed word2 executed OK

: word3 ( -- ) word2 ; <enter> word1 executed OK
word3 <enter> word2 executed OK
```

Figure 1

We create a first word, word1. Then we tell the system that word1 shall become a prelude of the next created word. Finally we create the next word, word2.

Now, when executing word2, we get the response "word1 executed word2 executed",

indicating that word1 was executed as a pre-executed word (prelude) of word2.

We create one more word, word3, using word2. While the definition for word3 is compiled, the message "word1 executed" occures, because word1 is executed as a prelude of word2 before word2 is compiled.

Figure 2 shows the same example, using finale instead of prelude.

```
: word1  ( -- )  ."  word1 executed " ;   <enter>  OK

´ word1  finale   <enter>  OK

: word2 ( -- )  ."  word2  executed " ;   <enter>  OK


word2   <enter>    word2 executed  word1 executed   OK


: word3  ( -- )  word2 ;   <enter>   word1 executed  OK

word3   <enter>   word2 executed   OK
```

Figure 2

When word2 is executed, we now get the response "word2 executed word1 executed", because word1 is a finale now. While word3 is compiled, we get the message word1 executed, but now word1 is executed as a finale after word2 was compiled.

What are pre- and post-executed words good for ?

The concept of pre- and post-executed words was found, when looking for a simple time- and memory-efficient approach to implement aspects of context-oriented and object-oriented technics for small microcontroller systems. I still didn´t have the time and the need to think about other applications but I expect the concept of pre- and post-executed words might be usefull for other things too.

Now lets take a short look how pre- and post-executed words can be used to early-bind methods to data. See Figure 3 for that purpose.

The new data type ascii is introduced. First a vocabulary ascii is created, to hold all the ascii-related methods. Then methods to fetch, store and display and create ascii data are defined.

Figure 3

```
\ Creating a methods context for the new data type ascii .

vocabulary ascii                    ascii definitions decimal

\ Creating methods to fetch, store and display an ascii character

´ forth  finale    ´ c@  alias  @ ( a -- b )

´ forth  finale    ´ c!    alias    ! ( b a -- )

´ forth  finale      : ? ( a -- ) c@ emit ;

\ Creating the defining word for the ascii data type

´ forth  finale    : variable ( -- ) ( ib: name )
                                [´] ascii  prelude  create bl c,  ;
```

The new ascii method variable creates a byte variable and assigns the vocabulary ascii as a pre-executed word to it. So, an ascii variable will invoke the context ascii, before it´s executed or compiled, so that the vocabulary ascii will be on top of the vocabulary search order, before the outer interpreter of the Forth system makes the next dictionary search access.

An ascii variable reference will be normally followed by an ascii method. The interpreter will pick up the name and (hopefully) find it in the ascii vocabulary that´s on top of the search order.

If the word is not a member of the ascii vocabulary then it might be found in another voacabulary deeper in the serach order. That might be okay or it might be an error condition. It´s only a restriction of this simple example and can be solved better with little effort.

All ascii methods have got the vocabulary forth as post-executed word. So all methods will switch back from the ascii contect to the context forth after beeing executed or compiled.

Allways returning to forth is only a restriction of this simple example. It can be handled in a more general form.

Figure 4 showes a print out of the vocabularies forth and ascii , after the source code from Figure 3 was compiled and Figure 5 is a record

of a short session, using the new data type.

```
ascii words  <enter>

variable  ?  !  @  OK


forth words <enter>

ascii  finale  prelude  ... many other names follow here ...  OK
```

Figure 4

```
forth definitions decimal

ascii variable  test  <enter>  OK

test ?    <enter>   OK

char A  test !  <enter>  OK

test ?  <enter>  A  OK

test @ emit <enter>  A  OK
```

Figure 5

Well, that are the basics of implicit context switching based on pre- and post-executed words.

We could add some syntactic sugar, introduce a more sofisticated context switching, add record structures and inheritance and we would end up with a more or less full featured extension for context and object oriented programming. See Appendix 1 for a syntax example.

How to implement pre- and post-executed words ?

Implementing pre- and post-executed words is not very complicated but it depends very much on the inplementation details of a forth system and it´s necessary to recompile the kernel or to patch it.

The header creating word in the forth kernel must be redefined and the outer interpreter has to be modified.

Redefining the header creating word

A pre- or post-executed word can be assigned to another Forth word by giving the header of that Forth word an additional cell-sized code-pointer. Furthermore two free bits are needed in the header, to indicate, whether a word has a prelude or a finale.

Figure 6 gives an example, how an implementation could be done:

```
forth definitions decimal

variable ´prelude

variable ´finale

: header ( -- ) ( ib: name )
          align
          ´prelude @  ?dup if  ,  then
          ´finale   @  ?dup if  ,  then
           header
          ´prelude @  if  set-prelude-bit  then
          ´finale   @  if  set-finale-bit    then
          ´prelude off
          ´finale   off  ;

: prelude ( xt -- ) ´prelude !  0 ´finale !  ;

: finale ( xt -- ) ´finale ! 0 ´prelude !  ;
```

Figure 6

Modifying the outer interpreter

It´s the outer interpreter´s job, to figure out, whether a word, found in the dictionary, has a prelude or a finale and to pre- or post-execute it. Assuming, that the outer interpreter uses the words **compile,** and **execute** , to compile or execute a word, we have to replace this words by the new words **?compile,** and **?execute** shown in Figure 7:

```
forth definitions decimal

: ?compile, ( xt -- )
            dup prelude-bit-set?  if  prelude-xt-@  execute  then
            dup compile,
            finale-bit-set?  if  finale-xt-@  execute  then  ;

: ?execute ( xt -- )
           dup prelude-bit-set?  if  prelude-xt-@  execute  then
           dup execute
           finale-bit-set?  if  finale-xt-@  execute  then  ;
```

Figure 7

So, implementing pre- and post-executed words is a relatively simple task, but in many existing forth systems you will not have two bits available in the header structure.

In this case you can succeed by implementing pre-executed words only or post-executed words only. Then you´ll only need one bit and halve the code for the implementation.

This might also be attractive for small systems with limited memory.

Using pre- **and** post-executed words will make the implementation of object oriented extensions less complicated and more elegant, but everything can be done using only pre- **or** post-executed words.

I started with this reduced approach and still use it, but I prefere to have pre- and post-executed words supported, to have a higher degree of freedom for the further evaluation of the potential of the concept of pre- and post-executed words.

References

[Bro84] L. Brodie. Thinking Forth. Prentise Hall, 1984.

[Fors94] L. Forsley. Rhyme, Reason and the Tao of Forth. Proceedings euroFORTH94.

[Gas93] M.L. Gassanenko. Context-Oriented Programming: Evolution of Vocabularies. Proceedings euroFORTH93.

[Mat89] J. Matthews. FORTH applications in engineering and industry. Ellis Horwood Limited, 1989.

[Vac90] G.-U. Vack. Programmieren mit FORTH. VEB Verlag Technik, 1990

[Woe92] J. Woehr. Forth: The New Model. A Programmer´s Handbook. Prentise Hall, 1992.

[Mah97] M. Mahlow. Kontextorientierte FORTH Systeme. FORTH-Tagung 1997

[Schl98] K. Schleisiek. Prelude - Objekte und Methoden mit Fehlerprüfung.Vierte Dimension 3/98, Forth-Gesellschaft e.V.

Appendix 1 : Syntax examples for object-oriented programming, based on implicit context switching, based on pre- and post-executed words.

```
forth definitions decimal

class:  .byte           \ create the new class  byte
                        \ a class instance is created in the form :     .byte  object:  name
byte field              \ a .byte instance has a one byte data field

method  ´ c@  alias  @ ( oa -- b )              \  fetch a byte
method  ´ c!    alias   !  ( b oa -- )          \  store a byte
method                  : ?  ( oa -- ) c@  .  ;         \  display a byte


forth definitions decimal

class:  .char           \ create the new class  char
                        \ a class instance is created in the form :     .char  object:  name
.byte inherited         \ class .byte is inherited

method  : ? ( oa -- ) c@  emit  ;               \  overwrite inherited method


forth definitions decimal

class:  .short          \ create the new class  short
                        \ a class instance is created in the form :     .short  object:  name
2 bytes field           \ a .short instance has a two byte data field

method  ´w@  alias  @ ( oa -- w )               \  fetch a short
method  ´ w!    alias   !  ( w oa -- )          \  store a short
method  : ? ( oa -- ) w@  .  ;                  \  display a short


forth definitions decimal

class:  .point          \  create the new class  point
                        \  a class instance is created in the form :     .point object:  name

.point structure begin                          \  start of data definition
        .short field: x                         \  the x-coordinate of a point
        .short field: y                         \  the y-coordinate of a point
.point structure end                            \  end of data definition

method  : @ ( oa -- x y ) dup .point x @  swap .point y @  ;     \  fetch a points coordinates
method  : !  ( x y oa -- )  dup >r  .point y !  r> .point x !  ;     \  store a points coordinates
method  :  ?  ( oa -- ) .point @  ." x = " swap .  ." y = " .  ;     \  display a points coordinates
```

```
forth definitions decimal

class:  .rectangle      \ create the new class  rectangle
                        \  a class instance is created in the form       .rectangle object:  name

.rectangle structure begin
                .point field: ulc           \  the upper left corner
                .point field: lrc           \  the lower right corner
.rectangle structure end

method    \ display rectangle coordinates
: ?  ( oa -- ) dup ."  ulc: " .rectangle ulc ?  ."  lrc: " .rectangle lrc ?  ;


forth definitions decimal

class:  .colored-rectangle    \ create the new class  colored-rectangle
                              \  create an instance in the form    .colored-rectangle object:  name

.colored-rectangle structure begin
                    .rectangle inherited    \  the rectangle coordinates
                    .short field: color     \  the rectangle color
.colored-rectangle structure end

method    \  display rectangle coordinates and color
: ?  ( oa -- ) dup .colored-rectangle ?   ."  color: "  .colored-rectangle color ?  ;

forth definitions
```

Classes are special vocabularies. A class can inherit once. Methods can be compiled at any time and will be visible to all instances of that class and to all instances of the child classes of that class. The data structure of a class, once created, can not be extended later. The class must be redefined or a new class must be created to inherit and extend the older one.

On a 16 bit ANS Forth System it takes less then 200 bytes to implement pre- and post-executed words. It takes less then 400 bytes more, to implement an extension for context-oriented programming with a separate sealed search order for methods contexts (classes), that allready supports inheritance and it takes less then 900 bytes to allow object-oriented programming like shown in the syntax examples above.

You are invited to contact me in case of questions or if you are interested in implementation details, but be warned, I´m a very busy guy.