Vectoring Arrays of Structures

Rieks Joosten

State University of Utrecht
Laboratory for Experimental Physics
Princetonplein 5
Utrecht
the Netherlands

Hans Nieuwenhuÿzen

Sterrewacht "Sonnenborgh"
Zonneburg 2
Utrecht
the Netherlands

Abstract

Writing a program module requires the specification of the interface between this and other program modules (or the operating environment). Since modularity implies that you can replace one module by another, a mechanism is needed to switch the definitions (datastructures) that make up the program module interface. In this paper, a general way of switching the action of a set of Forth datastructures (which can be thought of as a program module interface) is presented.

Introduction

It is shown that software can be designed in a structured way, e.g. for switching hardware oriented drivers. Switching drivers can now be done in a very general and safe way. A general framework is defined, which is the (virtual) interface between the hardware driver and user software. When an actual driver is attached to this framework, its routines (fields) will be called by the user (program) through the VI (Virtual Interface).

Drivers generally need some special action before they can be used (e.g. initialization of hardware), or before they can be de-assigned (e.g. turning off power). The syntax allows such actions to be specified when defining the actual drivers. An example is given of a character I/O system, and a possible implementation is given based on FysForth vsn 0.3.

The technique can be used whenever a vector of routines is used, and when the complete (execution) vector needs to be switched.

In section 1, we describe a mechanism for interfacing between modules, using a block system built on different mass storage drivers as an example. In the second section we will deal in a more general way with problems that may occur, showing some examples of the syntax for building the structures that are introduced in the first chapter. Sections 3 through 5 deal with the syntax and semantics of all relevant structures in detail. The implementation used is given in section 6; section 7 shows the internal structure of the words that are built. After several examples in section 8, we discuss performance and tradeoffs. For those who are not familiar with the system on which the described mechanism is implemented, Appendix A gives the déviations from the used system with respect to the Forth '79 Standard.

1. Modular Programming

Modular programming is a programming technique that allows multiple programmers to work on the same job at the same time. This is achieved by assigning each programmer a part of the job, called a module. A detailed specification of the constraints to which the programmer has to work is provided: the interface between the module of his program and other defined modules. Therefore, modules have to be specified (e.g. what actions they can take), and the interfaces between modules have to be specified since modules must be able to communicate with other modules.

Forth system software includes chunks of software that could readily be written in terms of modules. As an example, let us see how one could write the 'block' words using program modules.

A block system contains among others the following words: 'BLOCK', 'BUFFER', and 'SAVE-BUFFERS'. These words use some kind of mass storage organization that is not important to a user: as long as he can efficiently use all the mass storage media that he wants to use, he may not be interested in the organization of blocks on the mass storage medium. When a system has to use different devices at the same time, there are some problems, like: 'How does the block system know which mass storage device is in use', and 'How does it know how to read from or write to this device?'

Here, modular programming can help you. Assume a mass storage program module that contains definitions that allow you to read a block from a mass storage device, write a block to it, etc. Now, some way to link both modules is needed: the module interface. The module interface contains a specification of the definitions the block system program module wants to use, but does not have available. The module interface should describe what definitions must be part of the mass storage driver program module.

Some kind of structure, being the equivalent of the module interface description, will have to be present in the block system program module so that you can define the whole of this module without having to worry about how the actual mass storage handling words are put together. This structure is called the 'Virtual *Mass Storage* Interface' (VMSI) structure, and contains the specifications of the actions that can be taken by a mass storage driver. The driver cannot do anything specific unless actual actions have been assigned to each of the words specified by this structure.

In the example, the following definitions are specified in the VMSI structure:

Definition type:	**Name:**	**Description:**
VARIABLE	ERR#	Error specifying number
CONSTANT	L#DEV	Nr. of blocks on the device
: or CODE	BREAD.....	Read a block from the device
: or CODE	BWRITE	Write a block to the device
: or CODE	RESET	Reset the device

Analogously, the actual mass storage driver program module has to contain a structure that will enable the correct coupling of the actual mass storage driver program with the block system program module, i.e. the VMSI structure. This structure (in the actual mass storage driver program module) will be called the 'Actual Mass Storage Interface' (AMSI) structure. So we have a VMSI and an AMSI that have to work together in actual use; together they can be thought of as being the interface between two modules. (The AMSI and VMSI structures are specific examples of the AI (Actual Interface) structure and the VI (Virtual Interface) structure, that will be introduced later).

Suppose we have available a mass storage device called DA:, and the following definitions:

DA:BREAD DA:BWRITE DA:RESET

that are incorporated in the AMSI structure; the routines in the block system module can use them through the VMSI structure.

Now suppose we have two mass storage devices, called DA: and DB:. The documentation that is supplied with these devices gives all the information needed to write the routines that read from, write to, and reset these devices. We also need to create a constant that contains the maximum number of blocks that fit on these devices, to create a variable that can be used by the read- and write routines to store error conditions into. For the purpose of the example, we will assume that the above definitions are also available for DB:, so now we have:

DA:BREAD DA:BWRITE DA:RESET and DB:BREAD DB:BWRITE DB:RESET

which read a block from, write a block to, and reset the devices DA: and DB: respectively. Obviously, both mass storage driver program modules will have an AMSI structure. The block system program module does not know which of the above modules it uses, since it only sees the VMSI structure through which it accesses the routines from the currently active AMSI structure.

Switching the currently active AMSI structure ensures that the block system program will from that moment on use another device. Suppose we have the routines 'DA:' and 'DB:' which will allocate the AMSI structure of the DA: or the DB: module respectively. First, after having loaded both the DA: and the DB: module and the block system program module, the VMSI from the block system progam module is not attached to either of the AMSI structures (see Fig. 1.)

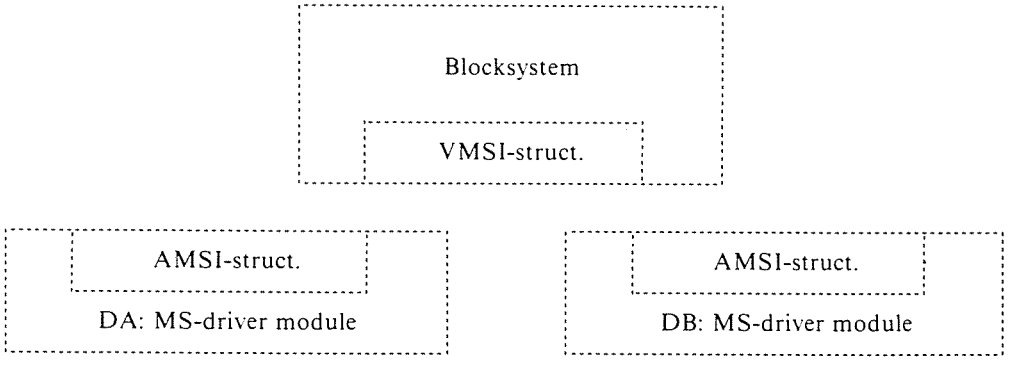


Figure 1. Organization after definition of all modules; VMSI and AMSI structures are not attached.

If at this time you try to use words from the block system, an error condition exists, because the block system words try to access some mass storage device through the VMSI structure which in its turn is not linked to any AMSI structure; the interface is incomplete.

When the word 'DA:' is typed, the AMSI structure of the DA: mass storage module is attached to the VMSI structure of the block systems module (see Fig. 2.). At this moment, executing words from the block system will use the definitions from the DA: module, which means that blocks are read from and written to the DA: device.

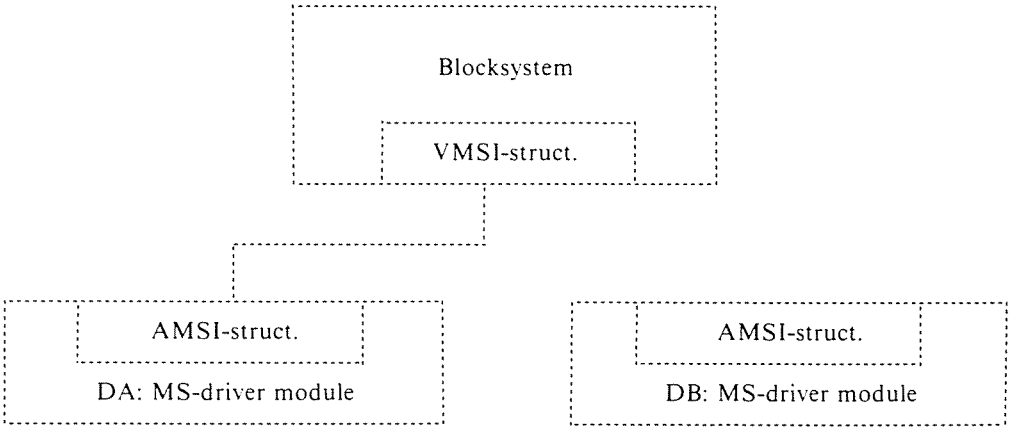


Figure 2. After typing 'DA:', the AMSI-structure of the DA: module has been attached to the VMSI-struct.

If now the word 'DB:' is typed, the AMSI structure of the DA: mass storage module is de-coupled from the VMSI structure of the block systems module. Then the AMSI structure of the DB: mass storage module is attached to the VMSI structure of the block systems module (see Fig. 3.). At this moment, definitions from the block system will use the routines of the DB: module, which means that blocks are read from and written to the DB: device.

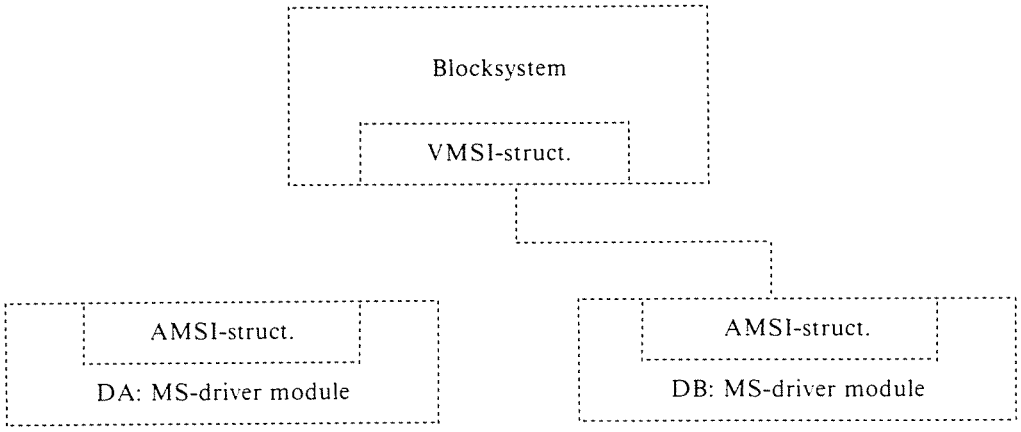


Figure 3. After typing 'DB:', the AMSI-structure of the DA: module is de-coupled. The AMSI-structure of module DB: is attached to the VMSI structure.

2. *Modular Interfaces*

We will now extend the example from the previous section and introduce a general syntax for interfacing two modules. One of these is the master module: it uses routines from the other module called the slave module. The master module contains the structure (the Virtual Interface (VI) structure) that specifies what routines should be in the slave module, and what these routines should do. The slave module also contains a structure (the Actual Interface (AI) structure), that takes care of the correct attaching of the slave to the master module. (Note that VI structure is the general structure, of which the VMSI structure was a specific example; the same holds for the AI and the AMSI structures).

The VI structures of the example of the previous chapter can now be defined. We assume here that the routines that are densely printed are available in the Forth system. See sections 3 and following for discussion of these words.

```
VIRTUAL   MS-DEVICE
   SVAR   ERR#
   SCONS  L#DEV
   SEXEC  BREAD
   SEXEC  BWRITE
   SEXEC  RESET
END
```

The definition of this VI structure specifies that an AI structure of type MS-DEVICE will consist of five fields, of which the first will act as a variable, the second as a constant, and the others as an executable Forth routine.

The AI structures for the mass storage driver program modules DA: and DB: are defined as follows, provided the words used are known to the Forth system.

```
ACTUAL     MS-DEVICE     DA:
                     0   INIT    ERR#
                  2002   INIT    L#DEV
 ' DA:BREAD       CFA    INIT    BREAD
 ' DA:BWRITE      CFA    INIT    BWRITE
 ' DA:RESET       CFA    INIT    RESET
END

ACTUAL     MS-DEVICE     DB:
                     0   INIT    ERR#
                  2002   INIT    L#DEV
 ' DB:BREAD       CFA    INIT    BREAD
 ' DB:BWRITE      CFA    INIT    BWRITE
 ' DB:RESET       CFA    INIT    RESET
END
```

This syntax has been chosen to help in defining the actions. Specifying syntax and semantics of words that together make up some concept takes careful analysis of what is going on, and what it really is that you want. This syntax may seem difficult at first because at first sight it does not resemble Forth. However, it was defined to do what it has been told to.

A few situations need careful analysis when installing or using this technique. If a master module is not attached to a slave module, (i.e. there is no currently active AI structure), execution of a word in the master module that uses any routine of the slave module, generates an error condition. Executing the word that attaches the slave module to the master module is essential for correct operation of the module.

Whenever a slave module is being attached to a master module, some initialization of the slave module can take place, e.g. when a disk driver module is attached to a block system module, the specified disk drive should be reset. This can be done by executing an 'OPEN' routine immediately after the coupling has been completed. Analogously, before de-coupling a slave module from a master module, some 'CLOSE' action may take place (e.g. by turning off the power of a device that consumes a lot of power).

Since there is no reason why a master or a slave module should be loaded before the other, this may give rise to 'forward references'. The system should take care that any currently active AI structure is properly 'CLOSED' and de-coupled when it is forgotten; if not, execution of routines from the master module may cause a system crash. This is a feature that most Forth systems cannot provide. However, the given implementation on the FysForth system does allow this, because it has a specific 'FORGET' action (ref. [2]).

The difficult part when using this technique is to find the correct set of interfacing routines.

3. *Syntax and Semantics of the Virtual Interface Structures*

There are two phases in defining interface structures:

1. define a Virtual Interface (VI) structure and
2. define an Actual Interface (AI) structure, together with the optional initialization of its fields.

We define the general syntax of a virtual structure:

```
VIRTUAL                <virtstruct>     \ Start the structure
                                        \ (Define the framework).
<n>*[  <$WORD>  <fieldname>  ]          \ Specify the fields
                                        \ in the framework.
END                                     \ End of structure
```

where:

- [....] is optional,
- < word > is a symbolic name,
- capital letters are existing names,
- <n>*[....] denotes that [....] occurs <n> times.
- <virtstruct> is the name of the routine that contains the general specification of structures of its type.
- <$WORD> is one of the following types:

  $CONS $VALUE $VAR $TEXT $EXEC

  Note that the words of type <$WORD> create Forth routines that act on the data in the respective field of the current actual structure. If there is no currently defined structure, an error occurs. How the routines act on the data in the current actual structure depends on how they were specified at creation time of <$WORD>. Additional routines of type <$WORD> can always be created.
- <fieldname> corresponds to the routine that acts on the contents of a named field in an actual structure.

4. *Syntax and Semantics of the Actual Interface Structures*

```
ACTUAL  <virtstruct>  <actualname>     Start the structure

[   OPEN>     <open routine> ]         Init. open-action
[   CLOSE>    <close routine> ]        Init. close-action

<m>*[  <p>*[              INIT"    <fld.name"> "<text>" ]
       <q>*[ <number> INIT    <fieldname>            ]   ]

    Initialization of (some of the) defined fields.

END                                    End of structure
```

where:

- [....] is optional,
- < word > is a symbolic name,
- capital letters are existing names,
- <n>*[....] denotes that [....] occurs <n> times.
- <virtstruct> is the name of the routine that contains the general specification of structures of its type. (see the definition of <virtstruct> on the previous page).
- <actualname> is the name of the routine that will 'close' the current structure of type <virtstruct> (i.e. finish), make itself current, and 'open' the current structure of type <virtstruct> (i.e. itself).
- OPEN> is the routine that assigns an <open routine> (which is the name of a Forth routine) to the general open routine for the actual structure <actualname>. The open routine for an actual structure is executed as soon as that actual structure is made the current one. Such routines handle the needed initialization.
- CLOSE> is the routine that assigns a <close routine> to the general close routine for the actual structure <actual name>. The close routine for an actual structure is executed as soon as that actual structure is released from the current structure. This can be achieved by assigning another actual structure as the current one (or by FORGETing the actual structure that happens to be the current one, assuming that FORGET is able to do such (see ref. [2]). Routines such as the <close routine> handle finalization.
- INIT" initializes the named field <fld.name"> using the string following the fieldname. This string should be delimited by quotes.

- <number> symbolizes Forth words and/or numbers that put one number on the stack. Example: 2002 returns a valid number, as well as the sequence: ' DEPTH CFA 2*
- INIT initializes the named field <fieldname> using the number on top of the stack.

5. *Action of the <$WORD>s*

The routines of type <$WORD> take care of the type of the fields in both virtual and actual structures. These routines have to contain information about:

- the length of the field (in bytes),
- the execution of the routines assigned to the field (<fieldname>), that will act on the data in that field.
- the initialization procedure INIT" on that same field, given the data storage address of the field on top of the stack, and a text in the inputbuffer.
- the initialization procedure of INIT on that same field, given a number 2nd on the stack and the data storage address of the field on top of the stack.

Note that two initialization routines are to be specified. An example of the necessity of this is when a $TEXT-field is to be initialized; it is possible that the address of the string that is to be assigned to the $TEXT-field is on the stack (in which case INIT would be used), whereas the text can also still in the input buffer, in which case INIT" will take care for the proper initialization of the $TEXT-field.

Creating routines of type <$WORD> is done as follows:

```
<length of field>
<cfa of <init> routine>
<cfa of <init"> routine>
<cfa of exec routine>          $WORD   <$WORDname>
```

where

- <length of field> is the length of the field (in bytes),
- <cfa of <init> routine> is the cfa of the routine that initializes the field starting at the address given on top of the stack, using the number 2nd on the stack.
- <cfa of <init"> routine> is the cfa of the routine that initializes the field at the address on top of the stack, given a string in the inputstream (the string is delimited with quotes).
- <cfa of exec routine> is the cfa of the Forth routine that acts on the contents of the address that is on the top of the stack.
- <$WORDname> is the name of the new <$WORD>.

Example of creating $WORDS: definition of $TEXT.

```
2                                   \ Length of a $TEXT-field

' ! CFA                            \ push cfa of INIT-routine

:ORPHAN HERE SWAP !                 \ push cfa of INIT"-routine
        &" WORD C@ 1+ ALLOT;

:ORPHAN @ ?DUP                      \ push cfa of DO-routine
        IF  COUNT TYPE
        THEN ;

$WORD $TEXT                         \ Create a new word of type
                                    \ $WORD with name $TEXT.
```

The word :ORPHAN is not standard: see Appendix A for its explanation.

6. *Implementation*

The implementation of the described structures has been done in FysForth vsn 0.3. This Forth system uses the TO-concept [1], and some defining words (which also solve some problems with forgetting) [2]. For system dependencies we refer to the FysForth vsns 0.2 / 0.3 User Manual [3]. The following code implements the described structures. Some routines use the reference word set as appended to the '79 standard. Appendix A of this paper contains a description of non standard words.

```
\ RIX/19830301
HEX                                     \ Set hexadecimal base

\ Helpvariables and helproutines
0  VALUE  VIRT/ACT                      \ See [1] for VALUE
0  VALUE  VADDR                         \ Generate VALUEs.
0  VALUE  STRUCLEN

: SINIS 0 TO VIRT/ACT                   \ Reset the values
        0 TO VADDR
        0 TO STRUCLEN ;

: =VIRT/ACT?                            \ Test for the wanted
        VIRT/ACT <>                     \ structure type; ABORT
        IF  CR ." "SYNTAX ERROR "       \ if the wrong structure
            ." ": VIRT/ACT = "          \ is found.
            VIRT/ACT . SINIS ABORT
        THEN ;

: @EXEC @ EXECUTE ;

\ Defining word for general (virtual) interface
DO>                                     \ See [2] for DO>, BUILD>
BUILD>          HERE TO VADDR           \ PRE-BUILD> etc.
                1 TO VIRT/ACT
                0 TO STRUCLEN
                0 ,                     \ Pointer to current struct
                0 ,                     \ Total structure length
PRE-BUILD>      0 =VIRT/ACT?            \ Check validity before
                                        \ building the Virtual.
DEFWORD>        VIRTUAL

\ Defining word for Fields in general interfaces
DO>             DUP @ @ ?DUP                \ <PFA>,<CURSTRUCT>
                IF   OVER 2+ @ +            \ <PFA>,<CURSTRUCT+OFFSET>
                     SWAP 4 + @ @EXEC       \ <CURSTRUCT+OFFSET> is on
                ELSE NFA COUNT 3F AND STYPE \ top of the stack
                     ." " NOT ASSIGNED. "   \ Abort when not
                     ABORT                  \ assigned.
                THEN
BUILD>          VADDR ,
                STRUCLEN ,
                DUP ,
                6 + @ +TO STRUCLEN          \ Adjust STRUCLEN with the
                                            \ length field of type <$WORD>
PRE-BUILD>      1 =VIRT/ACT?                \ Check validity before
                                            \ building the Field.
DEFWORD>        FIELD
```

```
\ Defining word for real (actual) interfaces
FORGET>         DUP >S                  \ Save pfa of actual.
                6 + S @ @ =             \ Is this actual current?
                IF   S 4 + @EXEC        \ Close the actual.
                     0 S @ !            \ De-assign the virtual.
                THEN -S                 \ Clean up system stack.
DO>             DUP >S @ @ ?DUP         \ Close current struc-
                IF   2- @EXEC           \ ture if it was current;
                THEN                    \ then switch the actual
                S DUP 6 + SWAP @ !      \ to become the current
                S> 2+ @EXEC             \ structure.
BUILD>          DUP ,                   \ Pointer to virt. struct
                0 ,                     \ Addr's of the default
                0 ,                     \ real open/close words.
                HERE TO VADDR           \ Save the field-addr
                2+ @                    \ Erase and allocate mem.
                HERE OVER ERASE         \ for the fields.
                          ALLOT
                2 TO VIRT/ACT           \ Set actual mode.
PRE-BUILD>      0 =VIRT/ACT?            \ Check validity before
                                        \ building the Actual
                [COMPILE] '             \ Put pfa of virtual on the
                                        \ stack.
DEFWORD>        ACTUAL

\ Routines for initializing fields in an ACTUAL structure
: (OPN/CLS)                             \ Initializes the fields
        2 =VIRT/ACT?                    \ in an actual structure
        [COMPILE] ' CFA                 \ that contain the OPEN
        SWAP VADDR + ! ;                \ and CLOSE routines.

: (INT) 2 =VIRT/ACT?
        [COMPILE] '                     \ Get pfa of the <FIELD> word.
        DUP @ VADDR 6 - @ ?PAIRS        \ Check congruence field/actual.
        DUP 2+ @ VADDR +                \ Calc. address to be init.ed.
        ROT ROT                         \ Calc. init. exec. address.
        4 + @ + @EXEC ;                 \ Initialize.

: OPEN>     -4 (OPN/CLS) ;              \ Initialize the OPEN field.
: CLOSE>    -2 (OPN/CLS) ;              \ Initialize the CLOSE field.

: INIT       4    (INT)    ;           \ Initialize using a number.
: INIT"      2    (INT)    ;           \ Initialize using a string.

\ The numbers (-4, -2, 4 and 2) that occur in the code of OPEN> CLOSE> INIT and INIT"
\ respectively are used by (OPN/CLS) and (INT) to find an address relevant for initialization
\ purposes; this is implementation dependent (see also the internal structure of ACTUALs and
\ $WORDS in section 7 of this article).

\ Completing the creation of a VIRTUAL Structure or Completing the creation and initialization
\ of an Actual Structure.
: END VIRT/ACT
    DOCASE 1 CASE STRUCLEN VADDR 2+ !   \ Completing a Virtual
      ELSE 2 CASE                       \ Completing an Actual
      ELSE          ABORT               \ Abort if neither
    ENDCASE SINIS ;                     \ Reset the helpvalues
```

```
  $WORDs (Fields specification words)
: NO"  CR ." "Wrong syntax used"                    \ Generate a syntax error
       ABORT ;                                     \ message.

DO>          [COMPILE] FIELD
BUILD>       ,                                     \ DO-routine.
             ,                                     \ INIT"-routine.
             ,                                     \ INIT-routine.
             ,                                     \ Length of field in an
DEFWORD>     $WORD                                 \       ACTUAL structure.

       2                                           \ Length of field = 2
' !           CFA                                  \ INIT-routine
' NO"         CFA                                  \ INIT"-routine
' @           CFA     $WORD $CONS                  \ DO-routine

       2                                           \ Length of field = 2
' !           CFA                                  \ INIT-routine
' NO"         CFA                                  \ INIT"-routine
' EXIT        CFA     $WORD $VAR                   \ DO-routine

       2                                           \ Length of field = 2
' !           CFA                                  \ INIT-routine
' NO"         CFA                                  \ INIT"-routine
' EXECTO      CFA     $WORD $VALUE                 \ DO-routine

       2                                           \ Length of field = 2
' !           CFA                                  \ INIT-routine
:ORPHAN -FIND 0= 4 ?ERROR CFA SWAP ! ;             \ INIT"-routine
' @EXEC      CFA     $WORD $EXEC                  \ DO-routine

       2                                           \ Length of field = 2
' !           CFA                                  \ INIT-routine
:ORPHAN HERE SWAP ! &" WORD C@ 1+ ALLOT ;          \ INIT"-routine
:ORPHAN @ ?DUP IF COUNT TYPE THEN ;                \ DO-routine
                     $WORD $TEXT
```

7. *Internal Structure of Words That Are Built*

This section describes the internal structure of the following word types:

<VIRTUAL>
<ACTUAL>
<FIELD>
<$WORD>

Note: within each type, every field is 2 bytes long, unless explicitly specified otherwise.

VIRTUALs

where:

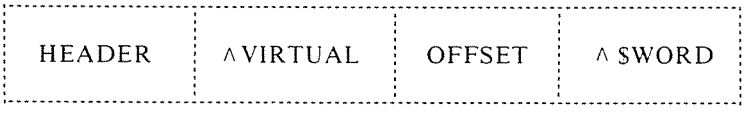
- HEADER is the link field, name field and code field of the routine;
- CURSTRUCT is the pfa of the current actual structure; if this value is 0, no current actual structure has been allocated.
- STRUCLEN is the length of the parameter field of an ACTUAL routine of this VIRTUAL type (in bytes). Since this length can only be known at the end of the creation of a complete VIRTUAL structure, this field is initialized by the routine END.

ACTUALs

where:

- ∧VIRTUAL is the pfa of the virtual structure that was used when this routine was defined (using ACTUAL);
- OPEN is the cfa of the routine that is executed immediately after this structure is made the current structure;
- CLOSE is the cfa of the routine that, when this structure is the current one, is executed immediately before this structure is de-assigned as the current one or when this structure is to be forgotten;
- FIELDS is a memory area with length STRUCLEN bytes, where
- STRUCLEN is the contents of the second field of the virtual structure that was used to define this actual structure.

FIELDs



where:

- ∧VIRTUAL is the pfa of the virtual structure that has been defined the latest;
- OFFSET is the offset in the field FIELDS within the actual structure where this routine finds its data;
- ∧$WORD is the pfa of the routine (of type $WORD) that was used to create the routine of type FIELD.

$WORDs

where:

- ∧EXEC is the cfa of the routine that acts on data in the field FIELDS of the current actual structure; the address of where this data can be found should be on the top of the stack;
- ∧INIT" is the cfa of the routine that initializes a field of type <$WORD>, given the address of the field on the stack and an initialization string in the input buffer;

- ^INIT is the cfa of the routine that initializes a field of type <$WORD>, given the address of the field on top of the stack and one (or more) number(s) below this address;
- LENGTH is the length of a field of type <$WORD>.

8. *Example of the Use of the Structures*

This section contains an example of the use of the technique described. Although designed for interfacing purposes, especially in conjunction with peripheral drivers, this technique may turn out to be useful in other applications.

Definition of the Virtual I/O Device

This example describes character I/O. In Forth, the terminal character I/O is organized using the words KEY and EMIT. Two additional helpful routines are ?KEY and ?EMIT (?KEY leaves a boolean telling whether a character is available from the current input device, and ?EMIT leaves a boolean telling whether the current output device can accept a character). For this example, the routine CURDEV prints the name of the current I/O device. The definition of such an I/O device is then described by:

```
VIRTUAL I/O-DEVICE                  \ Define a general I/O-device
                                    \ driver to contain:
      $TEXT     CURDEV              \ a textfield  called  CURDEV
      $EXEC     ?KEY                \ a routine    called  ?KEY
      $EXEC     KEY                 \ a routine    called  KEY
      $EXEC     ?EMIT               \ a routine    called  ?EMIT
      $EXEC     EMIT                \ a routine    called  EMIT
                                    \ Then terminate the definition
END                                 \ of the virtual I/O-device
                                    \ driver.
```

Definition of an Actual Terminal I/O Device

It is now possible to use the words CURDEV ?KEY KEY ?EMIT and EMIT in new routines. It does not matter how the actual routine of KEY etc. is accomplished as far as applications are concerned (this will only be of importance to the user who wants to specify his own I/O device). Assume that the following routines are available:

```
TT:?KEY        TT:KEY        TT:?EMIT        TT:EMIT
```

(which do a ?KEY function, a KEY function, a ?EMIT function and an EMIT function respectively for a given terminal). A terminal driver is then defined as:

Terminal driver 1.

```
ACTUAL I/O-DEVICE TT:               \ Create an actual instance of
                                    \ an I/O-DEVICE driver; in this
  INIT"  CURDEV    "Terminal "      \ case, a terminal called TT:
  INIT"  ?KEY      TT:?KEY          \ Initialize the textfield to
  INIT"  KEY       TT:KEY           \ contain the drivers name.
  INIT"  ?EMIT     TT:?EMIT         \ Attach the appropriate
  INIT"  EMIT      TT:EMIT          \ routines to the fields of
                                    \ type $EXEC.
END                                 \ Finish the initialization
```

An alternative way to obtain the same actual structure is:

Terminal driver 2.

```
ACTUAL I/O-DEVICE TT:

                    INIT"     CURDEV "Terminal "
' TT:?KEY   CFA     INIT      ?KEY
' TT:KEY    CFA     INIT      KEY
                    INIT"     ?EMIT     TT:?EMIT
' TT:EMIT   CFA     INIT      EMIT

END
```

The only difference between these two terminal drivers is in the definition of TT:; the one is initialized using INIT, while the other is initialized using INIT". Internally, there is no difference between the two TT:'s.

Specification of an Actual Terminal I/O Driver which has an Initialization Procedure.

Assume now that the terminal needs some initialization before it can be used, or that the user wants to set this terminal to a known state (e.g. erase the screen). Suppose the routine TT:OPEN is defined to do this. We now write:

Terminal driver 3:

```
ACTUAL I/O-DEVICE TT:                \ Create an actual instance of
                                     \ an I/O-DEVICE driver:
OPEN>  TT:OPEN                       \ Assign TT:OPEN as the routine
                                     \ that executes when the actual
  INIT"  CURDEV    "Terminal "       \ TT: is made current.
  INIT"  ?KEY      TT:?KEY           \ Init. the field that contains
  INIT"  KEY       TT:KEY            \ the name of the driver.
  INIT"  ?EMIT     TT:?EMIT          \ Initialize the fields of type
  INIT"  EMIT      TT:EMIT           \ SEXEC with the appropriate
                                     \ routines.
END                                  \ Finish the initialization
```

The sequence OPEN> TT:OPEN initializes the field in the actual structure that contains the execution address of the open-routine. This means that every time we type TT: , the routine TT:OPEN will be executed and thus initializes whatever is necessary.

Specification of an Actual Terminal I/O Driver which has a Termination Procedure.

In the same way as for the initialization procedure, a special action is taken when the current is de-assigned (i.e. when another I/O-DEVICE type routine is made current, thereby de-assigning the previous i/o device driver). Any special action to be taken before de-assigning an actual driver can also be specified. Suppose the routine TT:CLOSE will take such special de-assignment action for the terminal driver. We then have:

Terminal driver 4.

```
ACTUAL I/O-DEVICE TT:                \ Create an actual instance of
                                     \ an I/O driver.
CLOSE> TT:CLOSE                      \ Assign TT:CLOSE as the routine
                                     \ that executes when the actual
  INIT"  CURDEV   "Terminal "        \ TT: is de-assigned.
  INIT"  ?KEY     TT:?KEY            \ Init. the field that contains
  INIT"  KEY      TT:KEY             \ the name of the driver.
  INIT"  ?EMIT    TT:?EMIT           \ Initialize the fields of type
  INIT"  EMIT     TT:EMIT            \ $EXEC with the appropriate
                                     \ routines.
END                                  \ Finish the initialization
```

Specification of an Actual Terminal I/O Driver which has both an Initialization and a Termination Procedure.

Of course, when the terminal driver needs special action to be taken when it is made current, or when it is current and has to be de-assigned, the following syntax will be used:

Terminal driver 5.

```
ACTUAL I/O-DEVICE TT:                \ Create an actual instance of
                                     \ an I/O driver.
OPEN>  TT:OPEN                       \ Assign TT:OPEN as the routine
CLOSE> TT:CLOSE                      \ that executes when the actual
                                     \ TT: is made current. Assign
  INIT"  CURDEV   "Terminal "        \ TT:CLOSE as the routine that
  INIT"  ?KEY     TT:?KEY            \ executes when the actual TT:
  INIT"  KEY      TT:KEY             \ is de-assigned. Initialize
  INIT"  ?EMIT    TT:?EMIT           \ the field that contains the
  INIT"  EMIT     TT:EMIT            \ name of the driver, and the
                                     \ fields of type $EXEC.
END                                  \ Terminate initialization.
```

Defining Other Actuals of Type I/O-DEVICE.

Other types of actual I/O-DEVICE drivers can be defined once we know how the VIRTUAL I/O-DEVICE is defined. Assume that the routines INIT.PRINTER.PORT , LP:?EMIT and LP:EMIT have been defined to initialize the printer i/o, to see whether a character can be written to the printer, and to output a character to the printer.

Printer driver:

```
ACTUAL I/O-DEVICE LP:

OPEN> INIT.PRINTER.PORT

  INIT"  CURDEV   "Line printer "
  INIT"  ?KEY     0
  INIT"  KEY      ABORT
  INIT"  ?EMIT    LP:?EMIT
  INIT"  EMIT     LP:EMIT

END
```

Note that 0 is a regular Forth routine in most systems, and therefore can be used as the routine to initialize a field of type $EXEC. Analogously, a paper tape reader can be defined when the definitions INIT.PT.READER, PT.?KEY and PT.KEY have been defined (INIT.PT.READER initializes the paper tape reader hardware, PT.?KEY and PT.KEY perform the ?KEY and KEY action respectively on the paper tape reader):

Paper tape reader driver:

```
ACTUAL I/O-DEVICE PT.READER

OPEN> INIT.PT.READER

  INIT"   CURDEV    "Paper Tape Reader "
  INIT"   ?KEY      PT.?KEY
  INIT"   KEY       PT.KEY
  INIT"   ?EMIT     0
  INIT"   EMIT      ABORT

END
```

A separate papertape puncher driver can be defined as follows. INIT.PT.PUNCHER initializes the puncher hardware, PUNCHER.OFF turns the power off from the puncher, and PT.?EMIT and PT.EMIT perform the ?EMIT and EMIT action on the puncher.

Paper tape puncher driver:

```
ACTUAL I/O-DEVICE PT.PUNCHER

OPEN>  INIT.PT.PUNCHER
CLOSE> PUNCHER.OFF

  INIT"   CURDEV    "Paper Tape Puncher "
  INIT"   ?KEY      0
  INIT"   KEY       ABORT
  INIT"   ?EMIT     PT.?EMIT
  INIT"   EMIT      PT.EMIT

END
```

Suppose we want to read from the paper tape unit, but also to write to the puncher, e.g. for copying paper-tapes. Since both the paper tape reader and the paper tape puncher have hardware that has to be initialized, we have to write a routine that will initialize both the reader and the puncher hardware.

```
: INIT.PT                       \ Create a routine that will
        INIT.PT.PUNCHER         \ initialize both the papertape
        INIT.PT.READER ;        \ puncher and reader.
```

Now we are in a position to write the paper tape reader/puncher I/O-device driver:

Paper tape reader/puncher driver:

```
ACTUAL I/O-DEVICE PT:

OPEN>  INIT.PT                  \ Specify by an OPEN and a
CLOSE> PUNCHER.OFF              \ CLOSE action.

  INIT"   CURDEV    "Paper Tape Reader/Puncher "
  INIT"   ?KEY      PT.?KEY
  INIT"   KEY       PT.KEY
  INIT"   ?EMIT     PT.?EMIT
  INIT"   EMIT      PT.EMIT

END
```

9. *Performance and Tradeoffs*

Only one pointer has to be changed to switch ACTUALs in this particular implementation, which therefore can be very fast. The consequence however is that words of type FIELD have to fetch the address of the current 'actual' structure, add an offset to this address to obtain the address on which the execution routine can act, and to fetch the execution address. Therefore some extra calculations have to be done due to the extra level of indirection that is introduced.

High level printer drivers that were written to use this concept, were indeed significantly slower. Partly this is due to the execution overhead mentioned above. The other reason for being slower is that they were written in high-level Forth, whereas the earlier printer drivers were partly written in machine code. After rewriting the execution part of words of type FIELD in machine code, the new printer drivers were neither noticeably slower nor faster than the older ones. This demonstrates that there need not be a significant time overhead. Of course, this is implementation dependent.

An alternative implementation of the structures is that the switching of 'actuals' entails copying the parameter field of an entire ACTUAL structure to a fixed place in memory. This will result in a somewhat slower switch-action, but the words of type FIELD can execute faster. Whether this is actually needed for an application depends on the actual usage of the switch and execution routines. Note that this alteration of the implementation does not change the essence of this article! We cannot compare these implementations on practical experience.

Conclusions

Advantages of this concept are:

- ease of switching a set of routines.
- ability to define special action before and after the switching.
- possibility to work with 'virtual' definitions.
- ease of writing 'actual' definitions.
- ease of reading source code, implying good maintainability.
- possibility to optimize runtime routines within the structures, e.g. by using machine code definitions.

Disadvantages of this concept are:

- some small overhead in time; this overhead is implementation dependent.
- possible existence of problems due to the forward references used in the 'virtual' structures: most implementations of Forth systems cannot handle forward references safely: FORGETting such structures can lead to system crashes, unless precautions are taken. Using the implementation as described in this article, problems do not occur: FysForth vsn 0.3, which is used for this implementation, has been thoroughly tested in this area (also see Appendix A).

References

[1] P. Bartholdi: "The To Concept", Forth Dimensions, January 1979.

[2] R. Joosten and H. Nieuwenhuÿzen: "Ideas on a New Forget", FORML Proceedings, November 1981.

[3] R. Joosten *et. al.*, FysForth vsns 0.2/0.3 User Manual, State University of Utrecht, 1983.

[4] Proposal for a *standard* Forth interface with mass-storage using Forth blocks, Svend Lorensen and Jan Vermue, Book IV of the European Forth Users Group (EFUG), item 15. This is one of a set of papers mailed in April 1977 through the EFUG by H. Nieuwenhuyzen (Utrecht Observatory, the Netherlands), and redistributed by R. Milkey (Kitt Peak National Observatory, USA) in June 1977 for the U.S.

[5] W. Ragsdale, "A New Syntax for Defining Defining Words", FORML Proceedings, 1980.

Acknowledgements

We want to thank Thea Martin of the Institute for Applied Forth Research Inc. (Rochester), Larry Forsley and Carol Pruitt of the Laboratory for Laser Energetics (Rochester), Ted Bouk and Jerry Spitzner of the Eastman Kodak Company (Rochester), for having it made possible for one of us (Rieks) to stay in the United States during the last half of 1982. In this time, the first ideas of this Virtual/Actual concept were conceived. Discussions with several of the above mentioned people gave a good start toward its development.

We also want to thank Harm Braams and Frans Cornelis, for having helped to find the syntax currently used for the described structures, and for the time they spent analyzing, discussing this concept and its use, and for their proposals for enhancements.

Finally, we want to thank Dr. James Basile of the Long Island University, Greenvale, NY, for his comments on this paper.

Manuscript received June 1983.

Dr. Hans Nieuwenhuyzen received a Ph.D. in astronomy and physics from the State University of Utrecht in Utrecht, Holland in 1970. He is a senior research fellow at the University and promotes Forth for interactive environments. Dr. Nieuwenhuyzen is currently teaching courses in digital data processing including data transport, data base access and processing for applications in remote sensing, astronomy, high energy physics and medicine.

Mr. Rieks Joosten attended the State University of Utrecht concentrating in physics and is changing his concentration to informatiks. He is currently serving his military duty in Holland. Mr. Joosten is interested in operating system kernals and continues to use Forth and metaForth to develop easily generated, user friendly systems.

Appendix A: Definition of Non Standard Words Used

The Forth system that supports the implementation of the described structures is not a '79-standard system. This appendix describes any discrepancies between the '79-standard system and the words and concepts that are used in the examples. Some words refer to separate papers.

The Case statement.

The CASE statement used requires the following syntax:

```
<NR#1> <NR#2>   CASE   <EQUAL-PART>
                ELSE   <NOT-EQUAL-PART>
                THEN                          or

<NR#1> <NR#2>   CASE   <EQUAL-PART>
                THEN
```

where the code compiled by 'CASE' checks the top two numbers on the stack for equality. If the two top numbers are equal, both are dropped and the <EQUAL-PART> will be executed, after which execution continues behind THEN. If the top two numbers are not equal, only the number on top of the stack is dropped, and the <NOT-EQUAL-PART> is executed if present. After this, execution continues behind THEN.

The construction of the DOCASE-CASE-ELSE-ENDCASE statement is as follows:

```
<N> DOCASE
      <N1> CASE <<N>=<N1>> ELSE ( N is not equal to <N1> )
      <N2> CASE <<N>=<N2>> ELSE ( N is not equal to <N1> and <N2> )

    ( Expansion is limited to the implemented size of the stack )

    <NN> CASE <<N>=<NN>> ELSE ( N is not equal to <N1> and <N2> and ... <NN> )

    ENDCASE
```

'CASE' tests if the top 2 numbers are equal. If so, it drops both numbers and executes the words until an 'ELSE' or 'ENDCASE'. If not, it only drops the top number and skips to 'ELSE' or 'ENDCASE'.

The TO-concept.

The TO-concept [1] uses the following routines:

```
VALUE          TO          +TO          FROM
```

The New Forget: Creating Defining Words.

A new way of creating defining words [2, 5] is made possible by the following routines:

```
FORGET>      DO>      BUILD>      PRE-BUILD>      DEFWORD>
```

These words allow the building of defining words analogous to those built with CREATE and DOES>. However, special action can be taken by FORGET; such action is specified for all words of one type at the creation time of the defining word.

```
SYNTAX:
[ FORGET>          <FORGET-PART>        ]
[ RELOCATE>        <RELOCATE-PART>      ]
{ DO>              <DO-PART>          } { DOCODE>    <DO-PART>    }
  BUILD>           <BUILD-PART>
[ PRE-BUILD>       <PRE-BUILD-PART>     ]
  DEFWORD>         #<NAME>#
```

where:

- [....] is optional,
- < word > is a symbolic name,
- capital letters are existing names,
- <n>*[....] denotes that [....] occurs <n> times,
- | <1> | | <2 > | | <3> | ... | <m> | are a set of items <1> through <m>, of which one is required.
- <NAME> will be the name of a new defining word,
- <PRE-BUILD-PART> are those routines that are executed before a header will be built,
- <BUILD-PART> are the words that create a parameter field for a word of type <NAME>,
- <DO-PART> are the words that execute a routine of type <NAME>, with the parameter field address of the called routine on the stack,
- <RELOCATE-PART> would be the words that relocate a routine of type <NAME> with the parameter field address of the called routine on the stack.
- <FORGET-PART> are the words that execute before a routine of type <NAME> will be forgotten. The forget-part will have the parameter field of the routine-to-be-forgotten on the stack.

The minimum set consists of:

DO>	....		DOCODE>	....
BUILD>	....	or	BUILD>	....
DEFWORD>	....		DEFWORD>	....

The System Stack.

A special stack (system stack) is implemented for error recovery reasons. The routines that push an integer from the arithmetic stack onto this stack, do the reverse action, copy the top of the system stack onto the arithmetic stack, and delete the top integer from the system stack, are the following:

S S> >S -S

S - -> <INTEGER> copies the number on top of the system stack to the stack.

S> - -> <INTEGER> pulls the number from the top of the system stack onto the stack.

<INTEGER> >S - -> <> transfers an integer from the stack onto the system stack.

-S - -> <> drops the top integer off the system stack.

Other Non '79-Standard words.

Other words used in the article that are not part of the '79-Standard are the following:

?PAIRS CFA NFA :ORPHAN

?PAIRS

<INTEGER.1> <INTEGER.2> ?PAIRS - -> <>

checks whether <INTEGER.1> and <INTEGER.2> are equal. If they are not equal, ?PAIRS aborts, setting the value PAIR to <<byte1><byte2>>, where <byte1> and<byte2> are the low order bytes of the integers on top of the stack. This routine is used to check on program construction errors while using control structures.

CFA

<PFA> CFA - -> <CFA>

converts a parameter field address into its code field address.

NFA

<PFA> NFA - -> <NFA>

computes the Name Field Address of the routine whose Parameter Field Address is on top of the stack.

:ORPHAN ONLY WHILE EXECUTING

:ORPHAN - -> <ADDR> <SECURITY>

starts compilation of a colon routine, as ':', but without a name (HEADER); As it has no name, its code field address is left on the top of the stack below the security code. This code may be executed by EXECUTE.