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 deviations 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:

<table>
<thead>
<tr>
<th>Definition type:</th>
<th>Name:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARIABLE</td>
<td>ERR#</td>
<td>Error specifying number</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>L#DEV</td>
<td>Nr. of blocks on the device</td>
</tr>
<tr>
<td>: or CODE</td>
<td>BREAD</td>
<td>Read a block from the device</td>
</tr>
<tr>
<td>: or CODE</td>
<td>BWRITE</td>
<td>Write a block to the device</td>
</tr>
<tr>
<td>: or CODE</td>
<td>RESET</td>
<td>Reset the device</td>
</tr>
</tbody>
</table>

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:
Vectoring Arrays of Structures


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 program 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.
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
<n>*[ <SWORD> <fieldname> ] Specify the 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.
- <$SWORD>$ is one of the following types:
  \[ SCONS \quad SVALUE \quad SVAR \quad STEXT \quad SEXEC \]

Note that the words of type <$SWORD>$ 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 <$SWORD>$. Additional routines of type <$SWORD>$ can always be created.
- <field name> 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

\begin{verbatim}
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 <field name> ] ]

Initialization of (some of the) defined fields.
END End of structure
\end{verbatim}

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 `STEXT`-field is to be initialized: it is possible that the address of the string that is to be assigned to the `STEXT`-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 `STEXT`-field.

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

```
<length of field>
<cia of <init> routine>
<cia of <init"> routine>
<cia of exec routine>
SWORD <$WORDname>
```

where

- `<length of field>` is the length of the field (in bytes).
- `<cia of <init> routine>` is the cia of the routine that initializes the field starting at the address given on top of the stack, using the number 2nd on the stack.
- `<cia of <init"> routine>` is the cia 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).
- `<cia of exec routine>` is the cia 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 STEXT.**

```
2                     Length of a STEXT-field
' ! CFA               push cfa of INIT-routine
:ORPHAN HERE SWAP !   push cfa of INIT"-routine
&" WORD C@ !+ ALLOT:
:ORPHAN @ ?DUP        push cfa of DO-routine
    IF COUNT TYPE
    THEN :               Create a new word of type
SWORD STEXT            SWORD with name STEXT.
```

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

Helpvariables and helproutines
0 VALUE VIRT/ACT
0 VALUE VADDR
0 VALUE STRUCLEN

: SINIS 0 TO VIRT/ACT
    0 TO VADDR
    0 TO STRUCLEN :

: =VIRT/ACT?
    VIRT/ACT <>
    IF
        CR.
        "SYNTAX ERROR"
        ".": VIRT/ACT =
        VIRT/ACT. SINIS ABORT
    THEN:

: @EXEC @ EXECUTE :

Defining word for general (virtual) interface
DO>
BUILD>
    HERE TO VADDR
    1 TO VIRT/ACT
    0 TO STRUCLEN
    0 .
    0 .
PRE-BUILD>
    0 =VIRT/ACT?
DEFWORD>
    VIRTUAL

Defining word for Fields in general interfaces
DO>
BUILD>
    DUP @ @ ?DUP
    IF
        OVER 2+ @ +
        SWAP 4 + @ @EXEC
    ELSE
        NFA COUNT 3F AND
        "": NOT ASSIGNED.
        ABORT
    THEN

PRE-BUILD>
    VADDR .
    STRUCLEN .
    DUP .
    6 + @ +TO STRUCLEN

DEFWORD>
    FIELD
Defining word for real (actual) interfaces

```
FORGET>  DUP >S
       6 + S @ @ = IF S 4 + @EXEC
          0 S @ ! THEN -S

DO>     DUP >S @ @ ?DUP IF 2- @EXEC
          THEN S DUP 6 + SWAP @ !
          S> 2+ @EXEC

BUILD>  DUP . 0 . HERE TO VADDR
          2+ @ HERE OVER ERASE
          ALLOT
          2 TO VIRT ACT

PRE-BUILD> 0 =VIRT ACT?

[COMPILE] ' [COMPILE]

DEFWORD> ACTUAL
```

Routines for initializing fields in an ACTUAL structure

```
(OPN CLS) 2 =VIRT ACT?
[COMPILE] ' CFA
SWAP VADDR + ! :

(INT) 2 =VIRT ACT?
[COMPILE] ' [COMPILE]
DUP @ VADDR 6 - @ ?PAIRS
DUP 2+ @ VADDR + ROT ROT
4 + @ + @EXEC : 4

OPEN>    -4 (OPN CLS) :

CLOSE>   -2 (OPN CLS) :

INIT      4 (INT) :

INIT"     2 (INT) :

END VIRT ACT
```

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
ENDCASE SINIS :
```

Abort if neither
Reset the helpvalues
SWORDs (Fields specification words)

: NO" CR . " "Wrong syntax used"
  ABORT :

DO>  [COMPILE] FIELD
BUILD>  .
  .
  .
DEFWORD> SWORD

    2
'!  CFA
NO"  CFA
'@  CFA  SWORD SCONS

    2
'!  CFA
NO"  CFA
EXIT  CFA  SWORD SVAR

    2
'!  CFA
NO"  CFA
EXECTO CFA  SWORD SVVALUE

    2
: ORPHAN -FIND 0= 4 ?ERROR CFA SWAP ! : 
'@EXEC  CFA  SWORD SEXEC

    2
: ORPHAN HERE SWAP ! & " WORD C@ 1+ ALLOT : 
: ORPHAN @ ?DUP IF COUNT TYPE THEN :
  SWORD STEXT

Generate a syntax error message.

DO-routine.
INIT"-routine.
INIT-routine.
Length of field in an ACTUAL structure.

7. Internal Structure of Words That Are Built

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

<VIRTUAL>
<ACTUAL>
.FIELD>

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

VIRTUALs

HEADER  CURSTRUCT  STRUCLEN
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**

```
| HEADER | ^VIRTUAL | OPEN | CLOSE | FIELDS |
```

where:
- ^VIRTUAL is the pfa of the virtual structure that was used when this routine was defined (using ACTUAL);
- OPEN is the pfa of the routine that is executed immediately after this structure is made the current structure;
- CLOSE is the pfa 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

**FIELDSs**

```
| HEADER | ^VIRTUAL | OFFSET | ^SWORD |
```

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;
- ^SWORD is the pfa of the routine (of type SWORD) that was used to create the routine of type FIELD.

**$WORDs**

```
| HEADER | ^EXEC | ^INIT" | ^INIT | LENGTH |
```

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
STEXT CURDEV
SEXEC ?KEY
SEXEC KEY
SEXEC ?EMIT
SEXEC EMIT
END
```

Define a general I/O-device driver to contain:
- a textfield called CURDEV
- a routine called ?KEY
- a routine called KEY
- a routine called ?EMIT
- a routine called EMIT

Then terminate the definition 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:?EMIT TTEMIT
```

(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:
INIT" CURDEV "$Terminal"
INIT" ?KEY TT:KEY
INIT" KEY TT:KEY
INIT" ?EMIT TT:?EMIT
INIT" EMIT TT:EMIT
END
```

Create an actual instance of an I/O-DEVICE driver: in this case, a terminal called TT:

- Initialize the textfield to contain the drivers name.
- Attach the appropriate routines to the fields of type $SEXEC.
- 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
  '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:
  OPEN> TT:OPEN
  INIT"  CURDEV "Terminal 
  INIT"  ?KEY TT:KEY
  INIT"  KEY TT:KEY
  INIT"  ?EMIT TT:EMIT
  INIT"  EMIT TT:EMIT
END
```

Create an actual instance of an I-O-DEVICE driver:

Assign TT:OPEN as the routine that executes when the actual TT: is made current.

Init. the field that contains the name of the driver.

Initialize the fields of type SEXEC with the appropriate routines.

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 TT: IS DE-ASSIGNED.

INIT" CURDEV "Terminal " TT:CLOSE

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.
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

INIT.PT.PUNCHER
INIT.PT.READER : Create a routine that will
initialize both the papertape 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

[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.
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:

\[
\begin{align*}
\text{CASE} & \quad \text{EQUAL-PART} \\
\text{ELSE} & \quad \text{NOT-EQUAL-PART} \\
\text{THEN} & \quad \text{or}
\end{align*}
\]

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:

\[
\begin{align*}
\text{DOCASE} & \quad \text{CASE} \quad \text{ELSE} \quad \text{ENDCASE} \\
& \quad \text{CASE} \quad \text{ELSE} \quad \text{ENDCASE} \\
& \quad \text{ENDCASE}
\end{align*}
\]

`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:

\[
\begin{align*}
\text{VALUE} & \quad \text{TO} \quad \text{+TO} \quad \text{FROM}
\end{align*}
\]

The New Forget: Creating Defining Words.

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

\[
\begin{align*}
\text{FORGET>} & \quad \text{DO>} \quad \text{BUILD>} \quad \text{PRE-BUILD>} \quad \text{DEFWORD>}
\end{align*}
\]

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:

\[
\begin{align*}
[ & \quad \text{FORGET>} \quad \text{FORGET-PART}] \\
[ & \quad \text{RELOCATE>} \quad \text{RELOCATE-PART}] \\
[ & \quad \text{DO>} \quad \text{DO-PART}] \quad \text{DOCODE>} \quad \text{DO-PART}
\end{align*}
\]

\[
\begin{align*}
[ & \quad \text{BUILD>} \quad \text{BUILD-PART}] \\
[ & \quad \text{PRE-BUILD>} \quad \text{PRE-BUILD-PART}] \\
[ & \quad \text{DEFWORD>} \quad \text{NAME>}
\end{align*}
\]

\[
\begin{align*}
\text{SIGNALS} & \quad \text{DO>} \quad \text{DO-PART} \quad \text{DOCODE>} \quad \text{DO-PART}
\end{align*}
\]

\[
\begin{align*}
\text{SYNTAX} & \quad \text{FORGET>} \quad \text{FORGET-PART} \quad \text{RELOCATE>} \quad \text{RELOCATE-PART} \quad \text{DO>} \quad \text{DO-PART} \quad \text{BUILD>} \quad \text{BUILD-PART} \quad \text{PRE-BUILD>} \quad \text{PRE-BUILD-PART} \quad \text{DEFWORD>} \quad \text{NAME}>
\end{align*}
\]
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> BUILD> DEFWORD>
or
DOCODE> BUILD> 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 S  S>  >S  -S
S >  -> <INTEGER> copies the number on top of the system stack to the stack.
S>  S->  <->  <INTEGER> pulls the number from the top of the system stack onto the stack.
<INTEGER> >S  S->  <->  transfers an integer from the stack onto the system stack.
-S  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  S->  <->
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  S->  <CFA>
converts a parameter field address into its code field address.
NFA

\[ \text{\langle PFA \rangle NFA \rightarrow \langle NFA \rangle} \]

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

\text{\textbf{:ORPHAN}} \quad \text{\textbf{ONLY WHILE EXECUTING}}

\[ \text{\langle ORPHAN \rangle \rightarrow \langle ADDR \rangle \langle SECURITY \rangle} \]

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.