Appendix A

Communicating Novix NC4016s

At the start of this project we were given two Novix NC4016 boards. The intention being to work with these boards to provide systems that could communicate over multiple CPUs. The Novix NC4016 being a RISC based processor that is capable of executing a FORTH program at great speed.

A.1 Introduction

The Novix processor is sufficiently small that it takes only 4000 gates in a programmable logic array to implement. The chip has four different data paths, it is possible to use all four within one machine instruction, thus providing the capability of executing up to five FORTH instructions in one machine cycle. The majority of machine instructions are executed in one clock cycle. As a result a Novix system operating at 10 MHz has a peek effective throughput of 40 MHz or 40 MIPS (Million Instructions Per Second).

The machine instructions of the Novix are designed to execute FORTH. The FORTH systems supplied with the boards were designed to compile optimised native code for the Novix system.

In this annex we describe the Novix plug in boards in detail and describe how we made the two totally independent boards communicate with each other.

A.2 Programming

Four different implementations of the FORTH programming language were provided:

A.2.1 cmFORTH

This is an implementation by Chuck Moore. It is a basic system that has been modified to exploit the abilities of the Novix. We found the FORTH itself, whilst a valid implementation, to be lacking in features[1]. This is used in conjunction with the *SC*FORTH system to provide a programming environment on the Novix board.

A.2.2 *SC*FORTH

This is a system provided by Silicon Composers to operate on the IBM PC. It is a rather basic implementation of the language that had been extended to allow communication with the Novix boards.

[0] **This is a chapter taken from the Ph.D. thesis "Practical and Theoretical Aspects of Forth Software Development". A full copy of the thesis is available from the British library. Both the this chapter and the thesis are Copyright © Peter Knaggs.**

[1] This is a common feature of all of Chuck's FORTH systems.

Although the implementation is valid, it was of a rather basic system with few features.

To use the Novix board, the user would load the *SC*FORTH system. He would then load the **cm**FORTH system into a 16 KBytes dual ported memory buffer. The *SC*FORTH would then produce an interrupt signal that instructs the Novix to start executing the **cm**FORTH system. This provides a programming environment on the Novix system and a support environment on the IBM PC.

Having set the Novix executing the **cm**FORTH system, and any application the user may have loaded, he can then exit from the *SC*FORTH system back to the IBM PC's operating system. Thus leaving the Novix system to continue executing its application. This is only valid if the application does not require any facilities from the IBM PC "Host" (see section A.4), as these facilities have now been removed.

A.2.3 *Poly*FORTH

This is a fully implemented FORTH development system. Two versions of the system are provided. One operated on the IBM PC while the other operated on the Novix. This is a full FORTH system with a full screen editor, multi-tasking and meta-compilation. The system also provided a target compilation system for use with the Novix.

The Novix *Poly*FORTH system is also a full development system. It has multi-tasking, meta-compilation and optimisation. As with the Silicon Composers system, the user has to load the *Poly*FORTH system. This will then load the Novix *Poly*FORTH system into the dual ported memory and set the Novix executing it. The IBM PC system provides "Host" services for the Novix system.

As with the Silicon Composers system, once the Novix system has been loaded and is executing an application, the IBM PC's *Poly*FORTH system may be removed providing that the Novix application does not require any of the "Host" facilities provided by the IBM PC software.

A.2.4 FATWIN

The "*F*ORTH with *A*rgument records, Multi-*T*asking and *Win*dows" system developed by Bill Stoddart was supplied as a FORTH system that operated on the IBM PC. This is a full implementation of FORTH with many standard extensions and several non-standard features. Most of the extensions on this system are provided in a manner that is compatible with the *Poly*FORTH implementation.

Whilst this system was of little interest when the project began it was latter developed into the FORTH++ system and has played a large part in this project (see chapter 1).

A.3 Single Boards

We were using the *Poly*FORTH system to develop the multi-processor communications systems. This operated correctly with the first of the two boards but not with the second. The location of the dual ported memory of the first board was at segment `CC00` and segment `D000` for the second.

All of the software in the IBM PC *Poly*FORTH system used the variable `'FB` to hold the segment address of the dual ported memory. The software in the Novix *Poly*FORTH system does not need to know where this memory is located in the IBM PC, it simply uses the memory at `00000` though to `04000` as its communication area.

Thus we can change the board that the *Poly*FORTH system is communicating with simply by changing the value stored in the `'FB` variable. We defined two new words `FB1` and `FB2` to set the value of `'FB` to communicate with the first or second board as required. As the *Poly*FORTH system only copies the Novix *Poly*FORTH to the first board it is left to the user to "boot" the second board. This is done by selecting the second board and typing the FORTH word `BOOT`:

```
FB2 BOOT
```

Whilst this method of booting the second board works, we want to hide such information from the user. Thus, we redeveloped the code for `FB1`, `FB2`, `BOOT` (and `PLUG`) so that when the user selected a board that had not been initialised, the system would automatically copy the *Poly*FORTH system to the new board and set it operating. The code that created the words `FB1` and `FB2` was developed in such a way that the user can add new boards to the system at will. The code is given in section A.7.1.

A.4 Host Services

When the FORTH word `PLUG` is executed the *Poly*FORTH system connects with the current Novix board (indicated by the value of `'FB`). The `OPERATOR` task is set up to provide "Host Services" to the Novix system. The purpose of this is to provide the Novix system with a method of accessing the IBM PC's hardware. The *Poly*FORTH system provides access to the Keyboard, Video, Disks, Communication Ports and Printer.

The mechanism by which these services are provided is that of a "Mailbox". One mailbox is used for each device that the system is supporting. The mailboxes are stored in the dual ported memory, thus being accessible by both the Novix and the IBM PC. The mailboxes are of differing length depending on the device that they are controlling and the functions it can perform.

The Novix will place a value in the first word of the mailbox indicating the function it would like to perform. The IBM PC will take this value and use it as an offset into a table of functions[2] executing the required function. The IBM PC will place the value -1 in the mailbox to indicate to the Novix that it has completed the requested function. Whilst this is not a very good FORTH based approach to the problem, it is the way the original *Poly*FORTH system operated. In our work, we have tried not to alter the way in which the Novix communicates with the Host in any way thus we have inherited this message passing mechanism.

For the Novix to display a message on the IBM PC's screen, it would place the required text into the buffer area of the mailbox. It will then set a function code in the mailbox and wait for the IBM PC to acknowledge the function code. The IBM PC will interpret the function code and display the text in the buffer. Having completed the function it will send a message back to the Novix in the form of an acknowledgment. Having received the acknowledgment the Novix will now continue with its processing.

The system is designed in this way in order to provide multi-tasking on the Novix board. Let us say, for example, that two independent tasks on the Novix would like to access the same device. The first task would place its function code into the mailbox (and additional data if required). This task would then continue processing. However, the second task would see that the mailbox has a function pending (ie, that the function request code is not -1) and so will wait (in a multi-tasking manner) until the IBM PC acknowledges the first task request by placing -1 in the request code area. The second task can now continue and place its request in the mailbox.

The *Poly*FORTH "Host" system has a task constantly monitoring the function request value for each mailbox, with the exception of the Keyboard and Video mailboxes. When the IBM PC is `PLUG`ed into the Novix board the `OPERATOR` task monitors the Keyboard and Video mailboxes. One of the functions associated with this mailbox is "return to host". On receiving this request the `OPERATOR` reverts back to its normal action of interpreting keyboard commands for the IBM PC system.

A.5 Parallel Boards

The system described in sections A.3 and A.4 provides us with a mechanism that allows two (or more) separate Novix boards operating in parallel. There are still two problems to be solved:

1. Only one Novix board may have access to the "Host" facilities at any given time. This access is controlled by the operator connecting with the relevant board (via the use of `FB1` and `FB2`).

[2] This means that the function numbers must be even.

2. Although the boards may be operating in parallel, they are also operating in isolation. A method that allows the boards to communicate with each other is to be found.

We see two methods of solving the first of these problems. Namely the provision of "Host" services to all of the Novix systems available.

A.5.1 First Method

The monitoring of the mailbox function request area is performed by the word `AWAIT`. The code for `AWAIT` was redeveloped to scan for both Novix systems (see section A.7.2). Thus when the monitoring task executed the `AWAIT` word it would automatically monitor both of the Novix boards. When `AWAIT` returned the system would be ready to perform the required action for the correct board.

This system failed to operate to our expectations for various reasons:

1. The system is based around the ability of altering the value of the `'FB` variable to indicate the relevant Novix board. This is a *global* variable, thus, changing its value, will effect other tasks. As the system is multi-tasking, it is not possible to change such a variable and expect its value to be the same when next used.

   This is a common problem when dealing with multi-tasking systems. By changing the `'FB` variable from a *global* to a *user* variable, each task will have its own copy of the variable. Hence it can take on different values for each different task.

2. The code (as given in section A.7.2) is explicitly written for use on two boards and is not capable of expansion (without rewriting it each time a new board is added).

   By changing `MAKE.FB` we could make a cyclic list out of the possible Novix boards present. We are then able to rewrite `AWAIT` to scan through the cyclic list of possible boards. Hence, when the user creates another board, it would automatically be added to the list of boards. Thus, the monitoring software will automatically start servicing it.

   This has a problem in that the host will start servicing the board before it has been initialised. By moving the code that links the new board into this cyclic list to be the last function performed by the `FBOOT` word we stop the host from servicing the new board until after it has been initialised. The code for this system is given in section A.7.3.

3. The `DISKER` task provides access to the disk system, via use of the disk mailbox. This is a short mailbox holding two data fields, the value `#BLK` indicating which disk block the Novix requires and the value of `BUFFER` indicating whereabouts in the IBM PC's memory the block has been located. To read the contents of the disk block, the Novix issues keyboard input requests. The keyboard server will take its input from the indicated block rather than the keyboard.

   Whilst this is (in principle) the correct way of performing this function[3] it has two problems:

   (a) It relies on the interaction of two servicing tasks, the `DISKER` task (monitoring the disk mailbox) and the `OPERATOR` task (monitoring the keyboard/video mailbox). It is possible for the `OPERATOR` to be processing another board's request at the same time the `DISKER` request is made. However, the `OPERATOR` will see the flag indicating it should take input data from the disk block and will therefore pass the disk block information to the wrong Novix system.

   (b) It means that the Novix system can only process disk information when the keyboard/video mailbox server is in operation. Ie, the board in question is connected.

[3] Based on the description of disk accessing given in the Forth'83 standard.

If we were to "tighten up" the synchronisation between the two tasks, so that the `OPERATOR` task were to know for which Novix board the disk request applies, it would stop the first problem. However, the Novix system would still only be able to access the disk services when it has been connected too.

4. In order to communicate with the boards the user must connect with one of them (using the `FB1` or `FB2` words). This will set the `OPERATOR` task servicing the keyboard/video mailbox. However, as the keyboard is being treated as a normal device two (or more) Novix system may make a request to obtain input from the keyboard. One of the systems will be honoured, however, the user will have no knowledge as to which system he is communicating. This relates to the previous problem of the disk accessing being directed through the keyboard/video mailbox.

   By providing an additional task `TERM` that will service most of the keyboard/video mailbox requests, we can provide all of the boards with access to the video. We can also provide access to the disk system by synchronising the `TERM` task with the disk accessing of the `DISKER` task. This will leave the `OPERATOR` task to be used in the conventional manner.

   The `TERM` task will be able to process requests for text from the keyboard under two circumstances:

   (a) The Novix board making the request has previously made a request of the `DISKER` task (ie, is inputting a disk block).

   (b) The `OPERATOR` task had been directed to operate as the keyboard for the Novix system making the request.

   In this manner, we can provide a mechanism that allows any of the Novix boards to access the IBM PC's disk drives. The user will be able to `PLUG` into any of the boards and know that all keyboard communication is being directed to that board.

   This solution still has a problem. When the user is connected to one of the boards, it will stop any form of disk accessing. The Novix will issue a "Read data from the keyboard" request. While the `TERM` task is processing this request it will not be able to process any other requests. The solution to this problem is to totally redevelop the functions provided by the disk mailbox such that it operates independently of the keyboard mailbox.

A.5.2 Second Method

The second method of provided this facility is to totally redevelop the servicing code. By providing a single task that scanned and processed all of the mailboxes connected with the one board it is possible to add as many tasks as required, each servicing its own Novix system.

Whilst this method is conceptually simpler than the first, it means a complete redevelopment of the existing code, the converging of several tasks into one. Although this made the problems of handling the disk and keyboard into simple problems.

A.5.3 Comparison

Both methods provided the environment that is required. The first method is more difficult to understand and leads to complicated problems that were solved by redeveloping part of the server code (and the Novix code). It is simpler to use, as it is more automatic than the second method.

The second method is simpler to understand but requires a great deal more effort to totally redevelop the servicing code, although no Novix code has to be changed. The coding has to take into account the fact that several tasks may be requesting access to the same resource simultaneously whilst, in the first method, this is taken care of automatically (by the logical splitting of tasks).

The second method is more open to adaptability than the first. With the second method it is possible to use two entirely different FORTH systems on the Novix boards, each having its own servicing

task on the IBM PC. For instance, it is possible to run *Poly*FORTH on the first Novix board and **cm**FORTH on the second (provided that a servicing task has been developed to operate with the **cm**FORTH system). The first method is simply not easily extendible.

A.6 Communicating Systems

Section A.5 describes a system that allows any number of Novix boards to operate in parallel, accessing host services as required. The user may communicate with one board at a time from the keyboard. Although the boards may be operating in parallel, they are still unable to communicate with anything except for the host. In this section, we extend this system to allow different Novix boards to communicate with each other without the intervention of the user.

A.6.1 Hardware Restrictions

The main problem in providing such a facility is the Hardware Restrictions inherent in the design of the board. The boards provide a 16 KBytes area of dual ported memory that is mapped into the IBM PC's main memory area thus allowing the IBM PC to access the first 16 KBytes of the Novix memory. The exact location of this memory can be configured (in the IBM PC) by setting jump switches on the Novix board. No two Novix boards may have a configuration using the same memory area on the IBM PC for their dual ported memory as this would cause a "bus error", in addition to being a logical error. No two dual ported memory areas may overlap as this will also cause a "bus error".

Thus, for each Novix board in use this 16 KBytes of dual ported memory are physically separated (by being located on the Novix boards) and must be logically separated (by being mapped into different areas of the IBM PC's memory).

A.6.2 Communication

Given that the hardware design inhibits any form of inter-processor communication (except with the IBM PC), a software solution is to be found.

As the IBM PC is the only system that is able to communicate with all of the boards present, we can consider a facility to allow communication between boards to be an additional host service. As we have already modified the way in which the host services are provided we are able to extend the system to include inter-processor communication (between Novixs). For further information on host services see section A.4, and section A.5 for more information on their modification to allow multiple board operation.

Mailbox

We have provided an additional mailbox for "Inter-Processor Communication" (IPC). The mailbox has the format given in figure A.1.

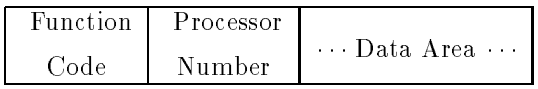

Figure A.1: Format of the IPC Mailbox

Where *Function code* is the function request code area, *Processor number* is a special data area used by all of the available functions and the *Data area* is where the IPC message is placed. The system we provide does not make any checks on the data area. The message may be in any format, as required by the application.

Identification

A system of identifying processors is required. There are basically three different ways of identifying a processor:

1. The segment address of its dual ported memory area can be given. This would seem the most useful option as it requires less code and is quick in operation. However, it means that each existing Novix will need to know the configuration of the other Novix's dual ported memory.

2. The address of its data area (in the cyclic list). This still has the problem that every Novix will have to have knowledge about every other Novix. This will also add a level of indirection to the system, thus slowing it down. In addition these addresses are likely to change as new code is developed.

3. An index value that indicates the Novix's position in the system. Thus the first system initialised is referred to as processor 1, the second as processor 2 etc. This requires two levels of indirection with extra processing by the IBM PC, however, it also means that the Novix systems needs no knowledge of the other Novix systems. Indeed it will also allow on-line reconfiguration of the Novix systems available.

   It would also be possible for an application to send messages using this IPC system between the IBM PC and the Novixs using this method. In such a case the IBM PC's identification number is given to be 0.

Functions

There are three functions associated with the IPC system:

Who am I? This function places the processor identification number of the calling Novix system in the *processor number* area of the mailbox, thus providing the method for a Novix to discover what its identification number is.

The number of Novix systems currently accessible to the host is placed in the *data area* of the mailbox. This is provided so that application software may configure itself to use all of the Novix systems available to the best advantage.

Wait will wait for a message being directed to this board. A task will be set up on the Novix system that waits for a message from another board. On receiving a message, it will interpret it and act on it as required. The processor identification number of the sending processor is placed in the *processor number* while the message is placed in the *data area*.

Message sends a message to another system. The message to be sent is placed in the *data area* while the processor identification number of the destination processor is in the *processor number* area.

The exact form of the message is application dependent. You may transfer data, in any form, from one processor to another (including the host). The host's message passing service simply copies the data area of the message buffer from one mailbox to the other. The target system must be waiting for a message otherwise an error code is returned (a -1 is placed in the "processor number" area).

As a board can discover its own processor number (via the *Who am I?* IPC function), it would be possible for a multi-tasking application to post a message to another task on the same board by placing its own processor id into the *processor number* field.

A.7 Code

In this section we present the code that was developed during this project. The code is presented in the order it was written with the multi-board boot code first followed by the first attempt at providing host services for multiple boards.

This code is explicitly designed for use with two boards. The redeveloped (true multi-board) version of the code is given in the final section.

A.7.1 Multiple Boards "Boot" code

We start by defining the new version of the two words `PLUG` and `BOOT`. Finally, we define the word `MAKE.FB`. We tell the system that a new board is present by defining a new data area for it with the `MAKE.FB` word. This defines a new word and defines a data area for the new board.

When the new word is executed (the board is referenced) for the first time it will load the *Poly*FORTH system into the board's dual port memory and start the board executing. `MAKE.FB` configures the new word's data area to execute the `FBOOT` word to perform this task.

The last action of the `FBOOT` word is to alter the action of the new word, so that next time it is executed, the system will execute the `FPLUG` word, thus given access to the given board.

We define a new version of `PLUG` that takes the address of the boards data area off the stack and ignores it.

```
: FPLUG ( addr -- ; Version of PLUG invoked by MAKE.FB words )
  DROP            \ Ignore data area address
  PLUG            \ Plug into the given board
;
```

Next, we define a new version of `BOOT` that takes the address of the boards data area and boots the given board. It will then change the value of the board's data area such that the next time the word is executed it will invoke the `FPLUG` word rather than the `FBOOT` word.

```
: FBOOT ( addr -- ; Version of BOOT invoked by MAKE.FB words )

  BOOT            \ Copy PolyFORTH to the Novix board

  ['] FPLUG       \ Get the execution token for FPLUG
  !               \ Alter the data area of the MAKE.FB word

  PLUG            \ Connect with the Novix board
;
```

Now we can define the `MAKE.FB` word. This is a creating word, it will create a named reference for the board, the dual ported memory of which is located at the given segment address.

When the new word is executed the system will connect with the given board, booting it, if the board has not been accessed before.

```
: MAKE.FB ( seg --   ; Create Novix Board access word with
                       dual ported memory at seg )

  CREATE              \ Create the new word

  ,                   \ Its body has the segment address of the dual
                      \     port memory for the given board

  ['] FBOOT ,         \ And the execution token of the FBOOT word

  DOES> ( addr -- )   \ When executed the new word will:

        DUP @         \ Fetch the dual port memory segment address
        'FB !         \ Store it in 'FB

        2+            \ Move to the execution token
        DUP           \ Get the execution token
```

```
        @EXECUTE     \ Execute the word (FBOOT or FPLUG)
;
```

It should be noted that when the **@EXECUTE** is executed the **'FB** variable has been set to the correct value for the board in question. The word **FBOOT** is passed the address of the execution token in the word's data area, this is so that it may change it. However, when changed the word **FPLUG** is also passed this value, so we must ignore it.

Finally we make the two boards known to the system.

```
HEX               \ input in Hexadecimal

CC00 MAKE.FB FB1  \ Define FB1 to connect with the first board,
                  \    with its dual ported memory at segment CC00

D000 MAKE.FB FB2  \ Define FB2 to connect to the second board,
                  \    with its dual ported memory at segment D000

DECIMAL           \ revert input back to Decimal
```

A.7.2 First attempt at providing Host Services

This section provides the code for the first attempt at providing host services for both boards (as described in section A.5.1). We start by defining two constants to hold the values of the dual ported memory for the different boards:

```
HEX                \ Numbers are in Hexadecimal

CC00 CONSTANT 'FB1 \ The first board is at segment address CC00
D000 CONSTANT 'FB2 \ The second is at D000

DECIMAL            \ Back to Decimal input
```

Now we redefine the **AWAIT** function such that it looks in the mailbox for both Novix boards alternatively. Examining the mailbox of the first board. If that is -1 (no function) it will then look at the mailbox of the second board, etc. When the value is not -1 there is a function pending. The **AWAIT** word will return with the function number. This matches the behaviour of the original **AWAIT** except for searching both mailboxes.

```
: AWAIT ( mailbox -- n ; Watch mailbox for function from Novix boards )
  -1                  \ Put a dummy function code on the stack

  BEGIN
    DROP              \ Drop the old function code (-1)
    PAUSE             \ Allow other tasks to run

    'FB @ 'FB1 =      \ Switch boards:
              IF 'FB2 \    If 'FB is pointing to the first board
            ELSE 'FB1 \       then set it to the second board
            THEN 'FB ! \      else set it to the first board

    DUP
    request FB@ DUP   \ Get mailbox function code
    -1 = NOT          \ True if function code <> -1

  UNTIL               \ Repeat loop until the function code <> -1

  NIP                 \ Remove mailbox address from the stack
;
```

A.7.3 Revised Boot and Host code

The code given in sections A.7.1 and A.7.2 is described in section A.5.1. This section also gives the problems encountered with this code and the modifications needed to overcome these problems. In this section we present this modified code.

In this code, we provide a cyclic list of Novix board data areas thus allowing us to access as many boards as are available. Otherwise, the code is similar to the code already presented.

In order to handle the cyclic list, we define a global variable (`'FB.HEAD`) to hold the address of the first entry in this list. We initially store a `0` in this variable, thus a `0` value is used to indicate an empty list. In order to overcome the problem of indefinite postponement, we also define the `'FB.LIST` user variable[4] (see the description of the `AWAIT` word on page 111 for more information).

```
VARIABLE 'FB.HEAD      0 'FB.HEAD !
   USER* 'FB.LIST      0 'FB.LIST !
```

We now redefine the `'FB` variable, changing it from a global variable into a user variable, thus each task will have its own copy of the variable.

```
USER* 'FB
```

We now provide the `FPLUG` word that will take the Novix board's data area and ignore it (for the same reason as given in A.7.1).

```
: FPLUG ( addr -- ; Version of PLUG invoked by MAKE.FB words )
  DROP            \ Ignore data area address
  PLUG            \ Plug into the given board
;
```

We now define the `FBOOT` word to work in much the same way as given in section A.7.1. However, having booted the board, it will then add the board to the cyclic list of available boards. If the list is empty it will make this entry link to itself thereby making a cyclic list.

```
: FBOOT ( addr --   ; Version of BOOT invoked by MAKE.FB words )

  BOOT              \ Copy PolyFORTH to the Novix board

  DUP 4+            \ Move to the execution token area
  [']  FPLUG !      \ Alter to the FPLUG execution token

                    \ Add this data area to the cyclic list
  'FB.HEAD @ 0=     \ Is the list empty ?
  IF   DUP !        \ Yes => Make this the last entry of the list
  ELSE DUP          \  No => Copy head of list to this entry
       'FB.HEAD @ !
  THEN 'FB.HEAD !   \ Make this entry the head of the list

  PLUG              \ Connect with the Novix board
;
```

We now define the `MAKE.FB` defining word. The data area holds space for the link required by the cyclic list, the segment address of the dual ported memory and the execution token of the word to execute (`FBOOT` or `FPLUG`) when the new word is executed.

[4] It should be noted that the word `USER*` is a special version of the standard word `USER` which automatically allocates the next free word in the user area.

```
: MAKE.FB ( seg --   ; Create Novix Board access word with
                       dual ported memory at seg )

  CREATE             \ Create the new word

  0 ,                \ Its body has an initial link value of 0
  ,                  \ The segment address of the dual port memory
  ['] FBOOT ,        \ And the execution token of FBOOT

  DOES> ( addr -- )  \ When executed the new word executes:
        DUP 2+ DUP   \ Move to the segment address value
        @ 'FB !      \ Store the boards segment address in 'FB
        2+           \ Move to the execution token
        @EXECUTE     \ Execute the word (FBOOT or FPLUG)
;
```

It should be noted that when the `@EXECUTE` is executed the `'FB` variable has been set to the correct value for the board in question. The memory address of the start of the data area is placed on the stack before invoking the word `FBOOT` or `FPLUG`.

We should now make the two boards known to the system.

```
HEX                \ input in Hexadecimal

CC00 MAKE.FB FB1   \ Define FB1 to connect with the first board,
                   \    with its dual ported memory at segment CC00

D000 MAKE.FB FB2   \ Define FB2 to connect to the second board,
                   \    with its dual ported memory at segment D000

DECIMAL            \ revert input back to Decimal
```

We are now in a position to redefine the `AWAIT` word. This version of the word will look at the mailbox associated with each of the boards in the cyclic list. If the function code is -1, it will move on to the next board in the list, otherwise it returns the function code, setting up the `'FB` variable for communication with the board in question.

In order to overcome the problem of indefinite postponement, the word will always start scanning from the entry indicate by the `'FB.LIST` variable. When a function code is found, a pointer to the next entry in the cyclic list is placed in the `'FB.LIST` variable. The word will start scanning from the next entry in the list when next invoked thus removing the possibility of indefinite postponement.

When the word is first invoked, the value of `'FB.LIST` will be `0` (uninitialised). The word will then check the global head of list (`'FB.HEAD`). If this is also found to be empty, the word will loop until such time as the list has an entry.

```
: AWAIT ( mailbox -- n ; Watch mailbox for a request from Novix boards )

  'FB.LIST @           \ Look at the local start of list

  BEGIN
    0=                 \ Loop if the list is empty
  WHILE
    PAUSE              \ Allow other tasks to run
    'FB.HEAD @         \ Get the global head of list
  REPEAT               \ Repeat until the list is NOT empty

  BEGIN
    PAUSE              \ Allow other tasks to run

    DUP 2+ @           \ Get this board's dual port segment address
    'FB !              \ Set the current board segment address
```

```
    OVER
    request FB@ DUP    \ Get the mailbox request
    -1 =

  WHILE                \ While the mailbox request is -1 (no request)
    DROP @             \ Move on to the next board in the list
  REPEAT

  SWAP @               \ Get the next list entry
  'FB.LIST !           \ Save for scanning next time

  NIP                  \ Drop the mailbox offset
;
```