MPE Forth 5

64180 Target

Warranties, copyright and licences

Warranty

MPE software products hold a warranty of 90 days. Software errors, reported within 90 days will be solved free of charge. After this time, fixes to problems are charged on a time and materials basis.

Technical support is available on the latest version of software. We do not maintain back-issues of software.

Modifications are only made to the latest version of the software. Therefore solving a problem may involve an upgrade to the most recent version.

Copyright

Make as many copies as you need for backup and security. The discs are not copy protected. Please treat this software like a book. It is copyrighted material and only one copy of it should be in use at any one time. If you need to photocopy the manual, you probably ought to purchase a second copy. Contact ourselves or your vendor for details of multiple copy terms and site licensing.

All the source files are copyright material and may not be further distributed without permission in writing from MicroProcessor Engineering.

The cross assembler in particular is copyright and the licence terms do not cover any use on target systems.

Licences

Any sealed object code generated by the cross compiler may be distributed without royalty. If your application is sealed (the user can't get at the Forth

and does not know it is written in Forth) there will be no licence problems - you are free to distribute the application.

If you leave the application open so that the text interpreter/compiler may be used, leave an MPE copyright message along with your own copyright notice as part of the sign on procedure, and contact MPE for details of OEM licensing terms. Please note that if the open Forth is not available to the end-user, but is **only for debug and test access**, then there is no royalty due to MPE.

OEM licences

The OEM licence allows developers to supply MPE documentation sets, and to use the ROM PowerForth utilities in their open Forth systems. Contact MPE for the current licence terms.

Registration

Because of the number of copies sold through dealers and purchasing departments we cannot keep track of all our users. If you fill out the registration form on the next page and return it or a photocopy to us, we will put you on our mailing list. This way we will be able to keep you informed of updates and new bolt-on goodies.

If you want direct technical support from us we will need these details to respond to you.

Software Registration Form

SEND TO:

MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO1 5AF
Hampshire
England
tel: (+44) 703 631441, fax: (+44) 703 339691

Customer details

Overseas customers - please include the country with your address.

Name:

Company:

Department:

Address:

Telephone number/extension:

Fax number:

Package details

Title: **MPE Forth 5 for 64180**

Processor and computer type:

Operating system:

Disc format: (3½, 5¼)

Serial number:

Purchase date:

Bought from:

Forth 5 for 64180

USER MANUAL

64180 Target

Version: 5.100

User Manual

Revision: 1.01

Date: 9 February 1994

Package No:	

For technical support:

Please contact your supplier

For further information:

MicroProcessor Engineering Limited
133 Hill Lane, Southampton
SO1 5AF, UK
Tel: 0703 631441
Fax: 0703 339691
Net:mpe@cix.compulink.co.uk

MPE Forth 5 for 64180

Microprocessor Engineering Limited
1993

Acknowledgements

MPE would like to thank the following people for all their involvement in the production of this product:

Jon Lee, Stephen Pelc, Paul Gallienne, Gary Ellis

Microprocessor Engineering Limited
133 Hill Lane
Southampton
SO1 5AF, UK

Table of Contents

Blank Page

List of Figures

List of Tables

Installing the system

It is recommended that you install the MPE Forth 5 64180 Development System by using the supplied installer. The installer helps you through the installation process and will make sure you have all the files you need.

System requirements

To install and use the development system you need:

- IBM PC or compatible with DOS version 3 or higher with 480Kbytes of available memory
- A hard disc with at least 1.5Mbytes of free disc space

Running the installer

The installer is supplied on issue disc #1.

To install the development system from drive A:, place the installation disc (disc #1) in drive A: and type

A:INSTALL

at the DOS prompt.

Selecting the installation drive

The installer lists all the available drives on your PC. Drive C: can be selected by pressing ENTER. If you want to install on a different drive, select a drive using the cursor keys followed by ENTER.

Selecting the installation path

The installation path is the path to the directory where the system is to be installed. Press ENTER to use the default path.

Standard or custom installation?

The installer asks you whether you require a standard or custom installation. Select standard to install the complete system. Select custom to chose which parts of the system you want to install. Your choice of standard or custom will normally depend on whether:

- you are a new user
- this is an upgrade
- you are adding features which you didn't install previously

A new user

If you are a new user and so are unfamiliar with MPE Forth development systems, you should install the complete system by selecting *standard*. This gives you the ability to explore what the development system has to offer.

An upgrade

If upgrading your development system, select *standard*. This installs the whole system as software versions may be incompatible.

Adding to the system

Select *custom* to choose which items to install. If you have previously installed only part of the development system, but you now want to install more of the system, select *custom*.

Standard installation

If you selected the standard installation, the installer installs the complete development system. It prompts for certain information:

- PC PowerForth Plus path
- The XShell path

It then prompts for the discs it needs.

Custom installation system

If a custom installation has been selected, the installer will prompt for certain information:

- The items to install
- The EPROM emulator driver required
- The EPROM emulator base address
- The PC Powerforth plus path
- The XShell path

The items to install

The installer needs to know what parts of the development system you want to install. By selecting YES for an item, the item will be installed. The space bar toggles between YES and NO.

The emulator driver

The development system is supplied with two drivers for the LeBurg EPROM emulator:

- TSR021
- TSR041

If you are going to use the LeProm emulator, select TSR021. If you are going to use the LeMeg or the LeBig emulators, select TSR041. If no EPROM emulator is going to be used, select *don't install a driver* .

The emulator base address

The installer needs to know what PC port address to map the emulator driver.

PowerForth path

PC PowerForth Plus is a Forth for your PC. Type the path where you want it to be installed. Press ENTER to use the default path.

XShell path

XShell is the cross compiler environment supplied as part of the development system. It is required to use the cross-compiler. Press ENTER to use the default path.

The MPE Development System

Now that you have installed the MPE development system, you may be wondering what you have got. The MPE development system is to the Forth-83 standard and consists of:

- XShell - the development environment
- the MPE Forth cross compiler with source
- source for generating a ROM target Forth
- source for generating an Umbilical Forth
- drivers for the LeBurg emulators
- PC PowerForth Plus

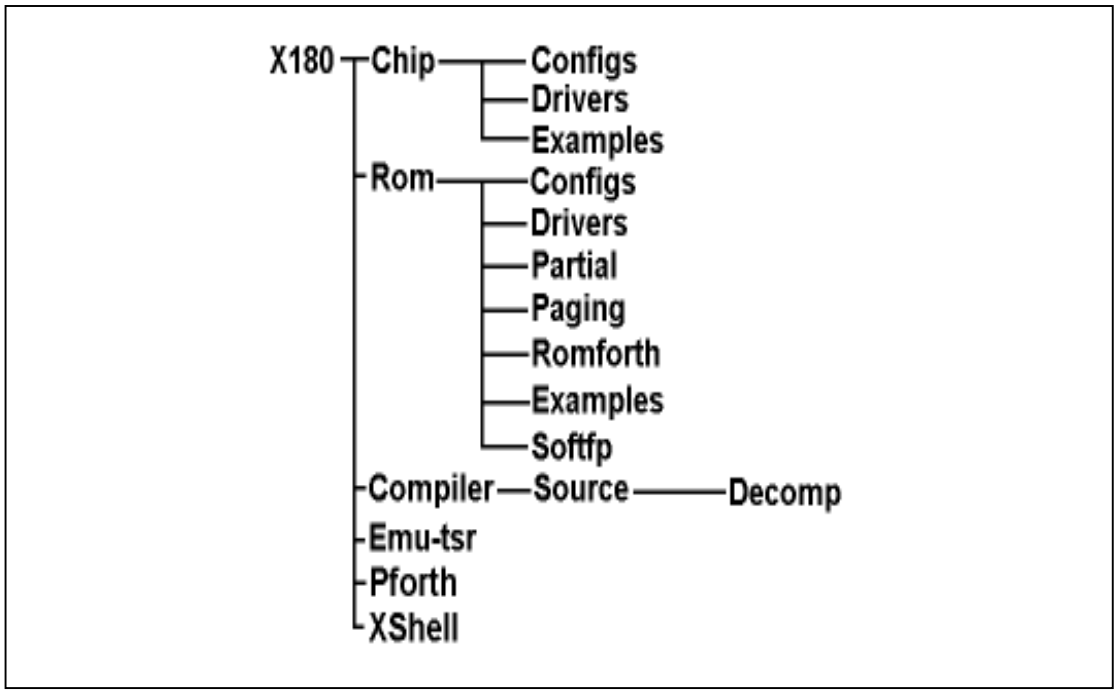
The installer creates, by default, the directory structure of figure 1. The places where XShell and PC PowerForth can be found may differ if the default directories were changed during installation.

XShell - the development environment

The MPE Development System is based around XShell. XShell is the environment used to:

- cross compile source code
- communicate with the target
- download the image to an EPROM emulator or programmer
- edit your source code
- run any DOS tools

XShell gives you a complete environment to generate, compile and execute code for your target board. For more detailed information see the XShell manual. The installer places XShell in the directory XSHELL.



□□□□□□□ 1 □ □□□ □□□□□□□□□□□□ □□□□□□□'□ □□□□□□□□□□ □□□□□□□□□□

MPE Forth cross compiler

The cross compiler can generate either a ROM target Forth or an Umbilical Forth from your source code. The source code for the cross compiler is supplied, so that you can extend the compiler and rebuild it from scratch if required.

The compiler can automate the generation of paged targets and also has a built-in cross-assembler. The compiler is in the directory COMPILER and the source is in the directories COMPILER\SOURCE and COMPILER\SOURCE\DECOMP.

ROM target Forth

Source code is supplied for developing a ROM target Forth. The Forth generated has a multitasker and software floating point.

It also has a larger wordset than an Umbilical Forth target, but is larger at 8K or more. If you require the multitasker, you must generate a ROM target Forth. The installer places the ROM target source code in the directory ROM. See chapter 3 on how to generate a ROM target Forth.

Umbilical Forth

Source code is supplied to generate an Umbilical Forth. Umbilical Forth is a significantly smaller Forth than the ROM target Forth. An interactive Umbilical Forth can be generated which is smaller than 2K. Umbilical Forth does not have all words defined in the ROM target Forth, but is useful if ROM space is at a premium. The Umbilical Forth source code is in the directory CHIP.

Leburg EPROM emulator drivers

The cross compiler can directly download code, as it is generated, to a LeBurg emulator. This is done via one of two TSR's:

- TSR021.COM - LeProm
- TSR041.COM - LeMeg and LeBig

These are in the directory EMU-TSR.

PC PowerForth Plus

PC PowerForth plus is a Forth for your PC. It can be used to prototype code in the host environment before porting to your target board. The installer places PC PowerForth plus in the directory \PFORTH.

Blank page

Generating a ROM target Forth

This chapter describes how to generate a ROM target Forth for your target board. It guides you through:

- setting up your hardware and software
- writing the serial line drivers
- modifying the memory map for your board
- compiling and running a target Forth

Supplied with your cross compiler are configurations for specific boards. If you have one of these boards, the generation of a target Forth is greatly simplified. If you do not have a supported board you will have to configure the cross compiler for your board and write new serial line drivers.

Is your board already supported?

If you have the GNC ue180 board you can use the supplied control file. By using this file, the installation of a ROM target Forth for your board will be greatly simplified. If you are using the GNC board then use the control file UE180.CTL. This is in the directory ROM\CONFIGS. There is also an example control file for the Z84C15 evaluation board.

If you do not have either of these boards you will have to modify a control file and serial line drivers for your board.

The control file

The control file contains all the details of your board that the cross compiler needs to know. These include:

- the memory map of your board
- whether you wish a log to be displayed
- the number of tasks in your system
- the clock rate of your board

As well as containing configuration information, the control file contains compiler directives and a list of files which are to be cross compiled. Once the cross compiler knows these items, it can generate a correct binary image from your source code. An example control file is shown in Appendix B.

Creating a control file

To create a new control file, copy an existing one and then modify it to match your board. This is normally easier than generating one from scratch. Example control files are in the directory ROM\CONFIGS.

The memory map

The memory map describes the addresses where ROM and RAM start and end in your target system. The memory map is defined by the:

- start of ROM
- start of RAM
- end of ROM
- end of RAM

From this information, the cross compiler places any items it needs in the correct area of memory.

Setting the memory map

The memory map is described in your control file, so once the file has been created, you can change the memory map definition to match your board.

The memory map is described in three parts:

- the start and end of ROM
- the start and end of RAM
- the end of memory

Setting the start and end of ROM

The start and end of ROM are defined by using the compiler directive **KERNEL**. **KERNEL** is used in the form:

rom-start rom-end KERNEL <name>

where rom-start is the address of the start of ROM, rom-end is the address of the end of ROM, and <name> is the name of the output file. The compiler automatically gives the filename <name> an extension .IMG so <name> must be just a name without an extension. The numbers rom-start and rom-end are, by default, in decimal, but can be entered in hex by preceding them by a $, e.g

$0100

The label <name> is also the name of the kernel page in a paged system. For more information see chapter 11, Paged targets.

Setting the start and end of RAM

The start and end of RAM are defined by using the directive, **KERNEL-RAM**. This is used in the form:

ram-start ram-end KERNEL-RAM

Setting the end of memory

The compiler needs to know the end of available memory. To set this use **MEM-END**, in the form:

xxxx MEM-END

where xxxx is the end of available memory. This area is where the stacks are placed.

Setting the compilation pages

In a non-paged system, the compiler must be instructed to compile into the pages defined by **KERNEL** and **KERNEL-RAM**. Therefore, after the memory map is defined you must code:

USE-CODE <name1>
USE-DATA <name2>

where <name1> is the name of the kernel ROM page defined with **KERNEL** and <name2> is the kernel RAM page defined with **KERNEL-RAM**.

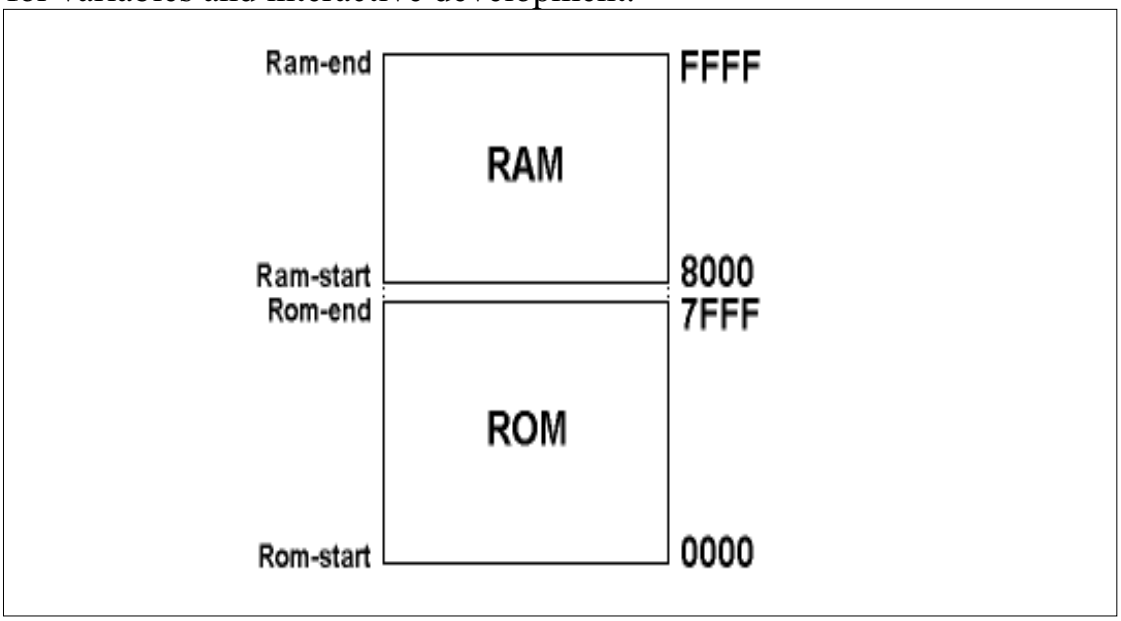
An example

If your target board has a memory map as in figure 2, your control file should be modified so that it reads,

```
$0000 $7FFF    KERNEL Rom180
$8000 $FFFF  0 KERNEL-RAM Rom180-Data
     $FFFF    MEM-END

USE-CODE Rom180
USE-DATA Rom180-data
```

This indicates two areas of memory with names ROM180 and ROM180-DATA. With this setup, your kernel will have 32k of ROM and you have 32k for variables and interactive development.



□□□□□□ 2 □ □□□□□□□□ □□□□□□ □□□

Modifying the serial line drivers

Your target board communicates with the external world via a UART. If you are using one of the more common external UARTs for the Z80/64180 family, or the internal ASCI on the 64180, one of the supplied serial drivers can be used. These are in the directory ROM\DRIVERS.

If you are using a UART for which code has not been supplied, you will need to write all the words required to:

- Initialise the UART
- Send a character
- Receive a character
- Test if a character has been received

All four words will normally be Forth **CODE** definitions. This is required so that the send and receive words are as fast as possible. Example serial line drivers in the files ROM\DRIVERS can be used as a template. As with the control file it is normally easier to modify an existing serial line driver file rather than creating your own from scratch.

Interrupt or polled drivers?

Two types of interrupt driver can be written:

- interrupt driven
- polled

Interrupt driven

An interrupt driven serial line can only be used if the UART generates interrupt signals when characters are received. An interrupt driven driver will allow buffered serial communications to be implemented with least processor overhead.

Polled

A polled driver will continuously poll a status bit in the UART to detect when the UART has either transmitted or received a character.

Initialising the serial line

The word **INIT-SER** must perform all the UART initialisation required. This includes setting:

- the baud rate
- any handshaking required
- the number of data bits
- the number of stop bits
- the parity to be used

It is recommended that the baud rate is initially set to 2400 baud until the target board is working. It can then be raised to 9600 or above, if possible, which makes a more responsive target.

Sending a character to the host

The target code needs to be able to send a character to the host for display. Therefore, you need to write a word which:

- waits for the transmit line to become available
- transmits a character to the host
- increments the variable **OUT**

The method used can be either a polled or interrupt driven driver but must be called **(EMIT)**. Once **(EMIT)** is written, it must be assigned to the defered word **EMIT**. The stack effect of **(EMIT)** is,

```
(EMIT)    \ char — ; send char to host
```

Receiving a character from the host

The target code needs the ability to receive a character from XShell. To do this it needs to:

- wait for a character to be received
- place the character on the Forth stack

The method used can be polled or interrupt driven but the word must be called **(KEY)**. Once **(KEY)** has been written, it must be assigned to the defered word **KEY**. The stack effect of **(KEY)** is:

```
(KEY)     \ — char ; wait for char to be received
```

Detecting a received character

The target needs **(KEY?)** to detect if a character has been received. This can be used as part of **(KEY)**. **(KEY?)** needs to:

- return true on the Forth stack if a character is available (-1)
- return false on the Forth stack if a character is not available (0)

Once **(KEY?)** is written, it must be assigned to the deferred word **KEY**. The stack effect of **(KEY?)** is:

```
(KEY?)    \ — t/f ; true if character received
```

Setting up the system

Setting up the system involves both hardware and software. The target board, PC, EPROM emulator/programmer and serial line have to be connected as well as configuring XShell to run the cross compiler.

Setting up the hardware

To generate an interactive Forth target you need:

- an IBM PC or compatible
- A serial cable
- A target board
- An EPROM emulator or programmer

Your PC needs to have at least one serial port for connecting to the target board, so making the Forth interactive. The serial cable should be connected to COM1 as this is the default port used by XShell. Other ports can be used by configuring XShell. See the XShell manual.

Setting up the software

To compile source code that generates a Forth target, you need to configure the cross compiler environment, XShell, to run the cross compiler. For detailed information on configuring XShell, see the XShell manual.

Running XShell

If during installation, you allowed the installer to modify your AUTOEXEC.BAT, then to run XShell you just need to type XS3. If you didn't or you haven't rebooted since you installed the system, then you need to state the full path of XShell. For example, the installer will place XShell in the directory, \X180\XSHELL by default.

Configuring XShell to use your control file

Before you can cross compile your source code, you must configure XShell. XShell requires the name of the control file you are using. The configuration file loads the remaining files so you need only to load the configuration file. To setup the configuration file as the file to be loaded,

i) run XShell while in the ROM directory

ii) type Alt-K, Configuration options

iii) press B, setup commands

iv) press E, compiler commands

v) type **ALL FROM-FILE** followed by the path and name of your configuration file and press enter, e.g.

ALL FROM-FILE CONFIGS\UE180.CTL

vi) press the escape key to return to the previous menu

vii) press E, save configuration

viii) Press the escape key to return to the host Forth

Your XShell configuration is now set to cross compile your configuration file.

Configuring the serial ports from XShell

XShell is used to communicate with the target. You therefore need to set up XShell to the same serial line settings that you are going to use on the target board.

To do this, type:

i) run XShell while in the ROM directory

ii) type Alt-K, Configuration options

iii) press D, serial line settings

iv) set up your settings by pressing letters a-z

v) press the escape key when finished

vi) type E, save configuration

vii) press the escape key to return to the host forth

Cross-compiling

Now the hardware and software have been set up, you can now cross compile the source code to generate an executable image.

Creating an image

To cross compile the source code, press F3. XShell clears the display and the cross compiler starts compiling. The compiler displays its sign-on message then compiles the source code.

The cross compile log

Following the compiler sign-on you see the cross compile log. As each word is compiled the compiler displays the word's address, its type and its shortened name. The type of item is coded as two characters as in table 1.

Turning the log on and off

Instead of having the data displayed for each compiled item, you can choose to only display a dot. The advantage of this is that the compiler spends less time displaying data and so the cross compilation is quicker. To do this, change the compiler directive in the control file from **LOG** to **NO-LOG**. The log can be turned on again by replacing **LOG** with **NO-LOG** in the control file.

Sending the log to a file

The cross compiler will redirect the log to a file instead of the display. To do this, use:

FILE: <name>

where <name> is the filename to generate. This directive must be placed before the command **CROSS-COMPILE**.

Sending the log to a printer

The cross compiler will send the log to a printer. To do this, use:

PRN:

before the command **CROSS-COMPILE**.

The compilation summary

Once the cross compiler has finished cross compiling the source code, it displays information about the compilation. This includes:

- any unresolved references
- the size of the compiled image
- the initialised RAM table address and length

Unresolved references are words which are referenced in the source code but are not defined. These can be due to spelling mistakes or not compiling some of your code.

The size of the compiled image is the amount of actual code output into the file. The actual file size will be the size of the ROM indicated by the memory map.

The RAM table is the place in ROM where a variable's initial value is stored. When the target board is reset, the initialisation copies this table into RAM. These initial values of variables will be modified in RAM when you store into a variable.

The created image

The image created by the cross compiler is a straight binary executable. It can be downloaded to a suitable EPROM emulator or programmer. The file has the name given when defining the memory map using the compiler directive **KERNEL**. It has the extension .IMG which cannot be changed.

Problems, Problems ...

If during compilation an error occurs, the compiler will stop and display the line on which the error occurred. The cross compiler shows the line number and the file name where the error occurred as well as the type of error that has occured.

Downloading the compiled image

Once the source code has been compiled the image needs to be downloaded to an EPROM emulator or programmer.

Downloading to a LeBurg EPROM emulator

The MPE cross compiler supports the LeBurg emulator. If you have a LeBurg emulator, the installer should have set up your XShell configuration to use it if it is already in the DOS path. In this case just press F4 and the LeBurg software should run. If the installer could not find your Leburg emulator software, you have to set up XShell to run your emulator software. Refer to the XShell manual.

Downloading to a different emulator

The binary image can be downloaded to any EPROM emulator as long as the emulator's software supports binary image files. Refer to the XShell manual on how to set up the XShell configuration and the emulator's software manual for download instructions.

Downloading to an EPROM programmer

The MPE development system supports the Sunshine programmer. If the installer found the programmer's software, then your configuration will be set up already. To run the programmer's software press F6. To set up XShell to use an EPROM programmer, refer to the XShell manual.

Running the target Forth

The image generated by the compiler has been downloaded to the target, it is ready to be reset and the Forth tested.

Switching to target mode

To receive characters from the target, XShell must be in target mode. The current mode is displayed on the top banner. If you are not already in target mode, type Alt-T or F5.

Reseting the target board

Once the image has been downloaded, you can reset the target board. You can either use the reset supplied on the board or if no reset is on the board, turn the board's power off and on again.

The sign-on

Once the board has been reset, the target should sign-on. You should see the message in figure 3. The version number and the number of bytes free will depend on your system. You should now have a working Forth. If the target didn't show the message, then you may have a problem with:

- the serial line drivers
- the memory map definition
- your target board
- your EPROM emulator/programmer

Each of these should be checked.

The serial line drivers

If you do not get the sign on message, your transmit word might not be working correctly. You can check that you can transmit a character up the serial line, by appending code for emitting a character up the serial line, onto the end of the initialisation word **INIT-SER**. Therefore a character can be transmitted and seen early in the initialisation sequence.

The memory map definition.

If the memory map for the ROM definition is wrong. The target may not sign-on at all. If the definition of the RAM memory map is wrong, the target may sign-on but may display ‘garbage’.

Your target board

It is always necessary to check the obvious. Is the serial line connected? Has your target board got power? EPROMs/RAM plugged in correctly? Are jumpers set correctly?

Your EPROM emulator/programmer

Check to see if your emulator is emulating an EPROM that your target board is expecting. If you have the wrong EPROM set, your target will not sign on.

Testing the Forth - an example

Once the Forth has signed-on, you need to test that it’s working properly. Type **WORDS** which will display all the Forth words available.

If this works then type in,

```
: FORTH-TEST                \ — ; A quick test for forth
  ." HELLO"
;
FORTH-TEST
```

This should display,

HELLO

followed by the ok prompt.

Cross-compiling an application

Once your Forth is working on your target board, you will now want to write and compile your application.

```
MPE 64180 ROM PowerForth v3.00
24373 bytes free

 ok
```

□□□□□□ 3 □ □□□ □□□□□□ □□□□□□□

Writing an application

Supplied with XShell is the TED editor. This can be run by pressing F2. A different editor can be used by changing the XShell configuration. See the XShell manual.

Modifying the control file

Once your application has been written, you can add it to the control file. Near the bottom of the control file, there is a list of commands in the form:

ALL FROM-FILE <name>

To compile your application files you add them to the end of the list.

Developing your application

As Forth is an interactive language, you can use this to your advantage by writing small sections of code and testing as you go. To help you do this, the ROM PowerForth utilities allow you to access your source files on the host. Your source files can be compiled from the target without cross compiling the whole application. See the chapter ROM PowerForth Utilities for more information.

Running your application

To compile the application you need to:

- run the cross compiler (press F3)
- download to the EPROM emulator/programmer (F4 or F6)
- reset the target

The target board signs-on. You can now test your application.

Generating a turnkey application

Once you have written your application, you will want to make it start when the target board is reset. This is known as a turnkey or autostarting application. Your application does not necessarily need to be interactive, so the compiler directive **NO-HEADS** can be used. This removes all the word headers, so making the final image more compact.

To make an application turnkey, use the directive **MAKE-TURNKEY** in the form:

MAKE-TURNKEY <name>

where <name> is the name of the word to run at startup. The word <name> must be defined before using this directive. The example in figure 4 generates a simple turnkey application when cross compiled. If you require the use of serial communications or the multitasker, you must initialise them in your application. To initialise the serial communications use the word **INIT-SER**. To initialise the multitasker use **INIT-MULTI**.

```
: MY-APP      \ — ;
  INIT-SER    \ Initialise the serial line
  BEGIN                       \ Application never ends...
    ." Hello"    \
  AGAIN                       \
;

MAKE-TURNKEY MY-APP
```

□□□□□□□ 4 □ □□□□□□□□ □□□□□□□□ □□□□□□□□□□□□□

Generating an Umbilical Forth target

This chapter describes how to generate an Umbilical Forth target for your target board. It guides you through:

- setting up your hardware and software
- writing the serial line drivers
- modifying the memory map for your board
- compiling and running a target Forth

Supplied with your cross compiler are configurations for specific boards. If you have one of these boards, generating a target Forth is greatly simplified. If you do not have a supported board you will have to configure the cross compiler for your board and write new serial line drivers.

Requirements for Umbilical Forth

To generate an interactive target you require:

- a LeBurg EPROM emulator
- interrupt driven serial drivers

If you want to define new words interactively, you need to use a LeBurg emulator. When the cross compiler generates code, it will write to the emulator. This normally 'upsets' the processor so the processor should be put to sleep while waiting for serial communications. Once the UART becomes available, the processor will be taken out of sleep mode and will continue processing.

Is your board already supported?

If you have the GNC ue180 board you can use the supplied control file. By using this file, the installation of an Umbilical Forth for your board will be

greatly simplified. If you are using the GNC board then use the control file UE180.CTL. This is in the directory CHIP\CONFIGS.

If you do not have this board you will have to create a control file and serial line drivers for your board.

The control file

The control file contains all the details of your board that the cross compiler needs to know. These include:

- the memory map of your board
- whether you wish a log to be displayed
- the clock rate of your boards crystal

As well as containing configuration information, the control file contains a list of files which are to be cross compiled.

Once the cross compiler knows these items, it can generate a correct binary image from your source code.

Creating a control file

To create a new control file, copy an existing one and then modify it to match your board. This is normally easier than generating one from scratch. Example control files are in the directory CHIP\CONFIGS.

The memory map

The memory map describes the addresses where ROM and RAM start and end in your target system. Therefore, there must be four items defined:

- the start of ROM
- the start of RAM
- the end of ROM
- the end of RAM

From this information the cross compiler places any items it needs in the correct area of memory.

Setting the memory map

The memory map is described in your control file, so once the file has been created, you can change the memory map definition to match your board.

The memory map is described in three parts:

- the start and end of ROM
- the start and end of RAM
- the end of memory

Setting the start and end of ROM

The start and end of ROM are defined by using the compiler directive **KERNEL**. **KERNEL** is used in the form:

rom-start rom-end KERNEL <name>

where rom-start is the address of the start of ROM, rom-end is the address of the end of ROM and <name> is the name of the output file. The compiler automatically gives the filename <name> an extension .IMG so <name> must be just a name without an extension. The numbers rom-start and rom-end are, by default, in decimal, but can be entered in hex by preceding them by a $.

The label <name> is also the name of the kernel page in a paged system. For more information see chapter 11, Paged targets.

Setting the start and end of RAM

The start and end of RAM are defined by using the directive, **KERNEL-RAM**. This is used in the form:

ram-start ram-end KERNEL-RAM <name>

Setting the end of memory

The compiler needs to know the end of available RAM. To set this use **MEM-END**, in the form:

xxxx MEM-END

where xxxx is the end of available memory.

Setting the compilation pages

In a non-paged system, the compiler must be instructed to compile into the pages defined by **KERNEL** and **KERNEL-RAM**. Therefore, after the memory map is defined you must code:

USE-CODE <name1>
USE-DATA <name2>

where <name1> is the name of the kernel ROM page defined with **KERNEL** and <name2> is the kernel RAM page defined with **KERNEL-RAM**.

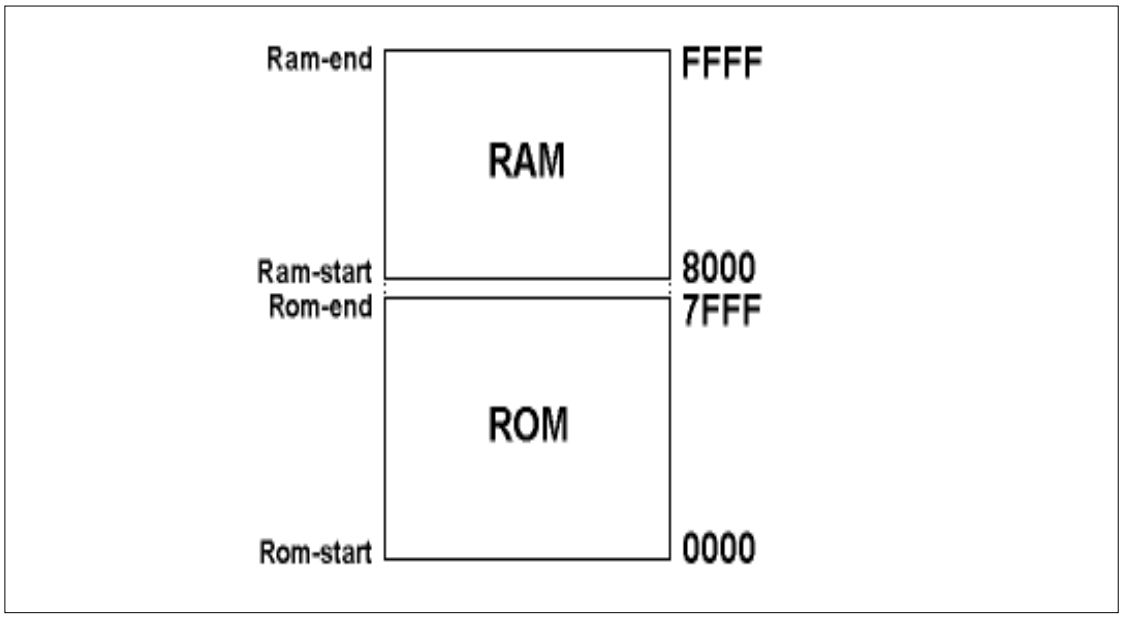
An example

If your target board has a memory map as in figure 5, your control file should be modified so that it reads,

```
$0000 $7FFF   KERNEL chip180
$8000 $FFFF 0 KERNEL-RAM chip180-data
$FFFF         MEM-END

USE-CODE Chip180
USE-DATA Chip180-data
```

This indicates two areas of memory with names CHIP180 and CHIP180-DATA. With this setup, your kernel will have 32k of ROM, and you will have 32K or RAM available for variables.



□□□□□□ 5 □ □□□□□□□ □□□□□□ □□□

Modifying the serial line drivers

Your target board communicates with the the external world via a UART. If you are using one of the common external UARTs used with the Z80/64180 families or the internal ASCI on the 64180, one of the supplied serial drivers can be used. These are in the directory CHIP\DRIVERS.

If you are using a UART for which code is not supplied, you will need to write all the words required to:

- Initialise the UART
- Send a character
- Receive a character
- Test if a character has been received

All of the four words will normally be Forth **CODE** definitions. This is required so that the send and receive words are as fast as possible. Example serial line drivers in the files CHIP\DRIVERS can be used for a template.

As with the control file it is normally easier to modify an existing serial line driver file rather than creating your own from scratch. The supplied drivers are in the directory, CHIP\DRIVERS.

Initialising the serial line

The word that must perform all the initialisation is the word **INIT-SER**. It must perform all the UART initialisation required. This includes setting:

- the baud rate
- any handshaking required
- the number of data bits
- the number of stop bits
- the parity to be used

It is recommended that the baud rate is initialily set to 2400 baud until the target board is working. It can then be raised to 9600 or above, if possible, which makes the target more responsive.

Sending a character to the host

The target code needs to be able to send a character to the host for display. Therefore, you need to write a word which:

- waits for the transmit line to become available
- transmits a character to the host.

The transmit word can either poll to detect whether the transmit line is available or, if available, an interrupt can be used. The word must be called **(EMIT)**. The stack effect of **(EMIT)** is,

```
(EMIT)    \ char — ; send char to host
```

Receiving a character from the host

The target code needs the ability to receive a character from XShell. To do this it needs to:

- wait for a character to be received
- place the character on the Forth stack

The receive word must be interrupt driven and the word must be called **(KEY)**. The stack effect of **(KEY)** is:

```
(KEY)     \ — char ; wait for char to be received
```

Detecting a received character

The target needs **(KEY?)** to detect if a character has been received. This can be used as part of **(KEY)**. **(KEY?)** needs to:

- return true on the Forth stack if a character is available (-1)
- return false on the Forth stack if a character is not available (0)

The stack effect of **(KEY?)** is:

```
(KEY?)    \ — t/f ; true if character received
```

Setting up the system

Setting up the system involves both hardware and software. The target board, PC, EPROM emulator/programmer and serial line have to be connected as well as configuring XShell to run the cross compiler.

Setting up the hardware

To generate an interactive Forth target you need:

- an IBM PC or compatible
- A serial line
- A target board
- An EPROM emulator or programmer

Your PC needs to have at least one serial port for connecting to the target board, so making the Forth interactive.

If the Leburg EPROM emulator is being used, you will also need to connect the emulator to the digital I/O card installed in your PC.

Setting up the software

To compile the source code that generates the Forth target, you need to configure the cross compiler environment, XShell. For more detailed information on configuring XShell, see the XShell manual.

Running XShell

If during installation, you allowed the installer to modify your AUTOEXEC.BAT, then to run XShell you just need to type XS3. If you didn't, then you need to state the full path of XShell. For example, the installer will place XShell in the directory, \X180\XSHELL by default.

Configuring Xshell to use your control file

Before you can cross compile your source code, you must configure XShell. XShell requires the name of the control file you are using. The configuration file loads the remaining files so you need only to load the configuration file. To setup the configuration file as the file to be loaded,

i) enter XShell while in the CHIP directory

ii) type Alt-K, Configuration options

iii) press B, setup commands

iv) press E, compiler commands

v) type **ALL FROM-FILE** followed by the path and name of your configuration file and press enter, e.g.

ALL FROM-FILE CONFIGS\UE180.CTL

vi) press the escape key to return to the previous menu

vii) press E, save configuration

viii) Press the escape key to return to the host Forth

Your XShell configuration is now set to cross compile your configuration file.

Configuring the serial ports from XShell

XShell is used to communicate with the target. You therefore need to set up XShell to the same serial line settings that you are going to use on the target board.

To do this, type:

i) run XShell while in the CHIP directory

ii) type Alt-K, Configuration options

iii) press D, serial line settings

iv) set up your settings by pressing letters a-z

v) press the escape key when finished

vi) type E, save configuration

vii) press the escape key to return to the host Forth

Cross-compiling

Now the hardware and software have been set up, you can now cross compile the source code which is automatically compiled down to your EPROM emulator.

Creating an image

To cross compile the source code, press F3. XShell clears the display and the cross compiler starts compiling. The compiler displays its sign-on message then compiles the source code.

The cross compile log

Following the compiler sign-on you see the cross compile log. As each word is compiled the compiler displays the words address, its type and its shortened name. The compiler type is coded as two characters as in table 2.

Turning the log on and off

Instead of having the data displayed for each compiled item, you can choose to only display a dot. The advantage of this is that the compiler spends less time displaying data and so the compilation is quicker. To do this, change the compiler directive in the control file from **LOG** to **NO-LOG**. The log can be turned on again by replacing log with no-log in the control file.

Sending the log to a file

The cross compiler will redirect the log to a file instead of the display. To do this, use:

FILE: <name>

where <name> is the filename to generate. This directive must be placed before the command **CROSS-COMPILE**.

Sending the log to a printer

The cross compiler will send the log to a printer. To do this, use:

```
PRN:
```

before the command **CROSS COMPILE**.

The compilation summary

Once the cross compiler has finished, it displays information about the compilation. This includes:

- any unresolved references
- the size of the compiled image
- the RAM table address and length

Words that are unresolved references are words which are referenced in the source code but are not defined. These can be spelling mistakes or some of the code is not being compiled.

The size of the compiled image is the amount of image downloaded to your emulator.

Problems, Problems ...

If an error occurs during compilation, the compiler will stop and display the line on which the error occurred. The cross compiler shows the line number and the file name where the error occurred as well as the type of error that occured.

Running the target Forth

The image generated by the compiler has been downloaded to the target, it is ready to be reset and the Forth tested.

Reseting the target board

Once the source code has been compiled and automatically downloaded to your LeBurg emulator you can reset the target board. Follow the instructions given by the cross compiler.

The sign-on

You will see the message in figure 7. The cross compiler itself displays this message, so the target is not necessarily up and working. To test the target board, you need to define a definition. Therefore if you type:

```
: FORTH-TEST                    \ — ; A quick test for forth
  ." HELLO"               \
;
FORTH-TEST
```

This should display,

HELLO

followed by the ok prompt.

If you didn't get this response, then you may have a problem with:

- the serial line drivers
- the memory map definition
- your target board
- your serial line
- your EPROM emulator/programmer

Each of these should be checked.

Cross-compiling an application

Once your Forth is working on your target board, you will now want to write and compile your application.

Writing an application

Supplied with XShell is the TED editor. This can be run by pressing F2. A different editor can be used by changing the XShell configuration. See the XShell manual.

Modifying the control file

Near the bottom of the control file, there is a list of commands in the form:

ALL FROM-FILE <name>

To compile your application files you add the files to the end of the list.

Running your application

To compile the application you need to:

- run the cross compiler (press F3)
- reset the target

The target board signs-on. You can now test your application.

Generating a turnkey application

Once you have written your application, you will want to make it start when the target board is reset. This is known as a turnkey or autostarting application.

To make an application turnkey, use the directive **MAKE-TURNKEY** in the form:

MAKE-TURNKEY <name>

where <name> is the name of the word to run at startup. The word <name> must be defined before using this directive. The example in figure 6 generates a simple turnkey application when cross compiled. To see the example working, you must switch XShell into target mode.

Blank Page

```
Umbilical Forth v3.00
Target: 64180
Copyright(C) 1993 Microprocessor Eng. Ltd.
BASE now in DECIMAL

  ok
```

□□□□□□ 7 □ □□□ □□□□□□□□□ □□□□□ □□□□□□□

```
: MY-APP      \ — ;
  INIT-SER    \ Initialise the serial line
  BEGIN                         \ Application never ends...
    42 (EMIT)  \
  AGAIN
;

MAKE-TURNKEY MY-APP
```

□□□□□□ 6 □ □□□□□□□ □□□□□□□□□ □□□□□□□ □□□□□□□□□□□

Optimising your target Forth

Once you have a target Forth, you may want to either reduce the size of your image or increase the execution speed of the code. This chapter describes those features of the MPE development system which help you with this aim.

Reducing the size of your image

During development you may need to reduce the size of your target image. For example, your application may have grown too large for your ROM space. Reducing ROM requirements is usually done by:

- removing headers
- factorising your code
- removing excess code
- using equates instead of constants
- using Umbilical Forth

Removing headers

To reduce the size of the compiled image, you can instruct the compiler to compile all or some of the code without heads. For each word defined, the cross compiler generates a header in the target image. A header is the name of the word as a counted string and is used when the target is used interactively. Therefore, by removing the heads of words you reduce the interactivity of your system.

Removing all headers

To remove the heads from all the code, use **NO-HEADS**. The compiler will produce code which will be greatly reduced in size, but cannot be used interactively.

Selectively removing headers

To select a number of words to be made headerless, use **INTERNAL** and **EXTERNAL**. **INTERNAL** instructs the compiler to stop generating headers, and **EXTERNAL** instructs it to generate headers again.

Factoring your code

When writing in Forth, code should be reused as much as possible. By reusing code, your target image can be reduced greatly. The smaller the procedures you use, the more easily they can be reused. In addition, small procedures are easy to test. Consequently code written with small procedures is normally more reliable.

Removing excess code

During development, debug and test code is inserted into the source. This code is easily left and forgotten about. By stripping out this excess code you can gain more space in the EPROM. A tool like MPE's cross-referencer, XRef, is invaluable for this sort of pruning.

Using equates instead of constants

An equate is a constant that just resides within the cross compiler. It therefore cannot be referenced when interactively debugging on your target system. The actual value of the equate is compiled 'in-line' instead of refering to a constant. Therefore you save the space on the target board for each constant (6 bytes + number of characters in the name) defined but sacrifice some interactivity. This only works if you don't refer to the equate many times, as an equate uses two more bytes than a constant, every time it is referred to.

Defining an equate

An equate is defined in a similar way to a constant:

xxxx EQU <name>

where xxxx is the value of the equate and **<NAME>** is its name.

Using an equate

An equate is used in the same way as a constant, by stating its name.

```
0100 EQU ADDRESS
ADDRESS 4 + EQU ADDRESS2

: SOME-WORD                \ —
 ... ADDRESS ... ;
```

Using Umbilical Forth

If you require a compact target Forth but without the inconvenience of removing target headers, you can use Umbilical Forth. Umbilical Forth gives you an interactive Forth in a very compact size (Umbilical Forth kernel is about 2k). The kernel doesn't contain all the words in the ROM target, so you might have to write a few words to get your code to compile or copy some code from the ROM target Forth. For more details see the chapter on Generating an Umbilical Forth Target.

Speeding up your code

The normal way to increase the speed of your code is to code strategic words in assembler. Good candidates for coding are:

- inner loops
- words containing a lot of stack manipulation words (**DUP**, **SWAP** etc)

Blank page

Z80/64180 Cross Assembler

The MPE cross compiler has a built-in cross-assembler. This gives you the ability to define new Forth words in assembler as well as in Forth. You can also assemble code to anywhere in memory.

Why write in assembler?

Forth is compact and quick, so why write in assembler? An assembler definition is normally quicker than a group of corresponding forth words.

Creating Forth words in assembler

Forth words can easily be defined in assembler. They increase the execution speed of your code and can sometimes make your code smaller.

Defining assembler words

Forth words written in assembler follow a similar form to a word written in forth. Instead of a colon you have **CODE**. Instead of semi-colon you have **END-CODE.** For example:

```
CODE <name>
  .   .
  .   .
   NEXT,
END-CODE
```

creates a word called <name>. Any assembler code between the **CODE** and **END-CODE** will be assembled into the word. When executed, the line **NEXT**, will stop the execution of the assembler and return to the calling word.

Note: If you do not have **NEXT,** your application will crash.

64180 register	Forth register	Function
BC	IP	Forth interpretive pointer.
SP	SP	Data stack pointer
IY	RP	Return stack pointer
IX		Used in the slow, but small implementation of NEXT,
	UP	User area pointer (in RAM)
HL		Points to CFA on entry to CODE definition.

Table 3 - The Forth registers

Writing assembler words

The syntax used for the opcodes have been kept as similar to that used by Hitachi as possible. See the list at the end of the chapter for a comparison of Hitachi versus Forth syntax.

Preserving the Forth registers

The Forth interpreter and compiler uses some of the target processor's registers. These must be preserved if they are used in the assembler. They can be saved on the stack, in memory or in other registers and restored at the end of the word. The 64180 registers that are used are shown in table 3. Note that in addition all registers of the alternate set are also used by the Forth itself in CODE definitions.

Executing an assembler word

A Forth word written in assembler is executed in the same way as a word written in Forth. It is executed by stating its name.

Assembling into memory

Assembler code can be assembled into memory which is not in a Forth word. To do this you need to:

- turn on the assembler
- write your assembler code
- turn off the assembler

To turn on the assembler, use the word **ASSEMBLER**. To switch back to Forth use the word **FORTH.** Between the **ASSEMBLER** and **FORTH** definitions, any assembler will be assembled. The assembled code will be placed in the dictionary without a header. The code can be executed by the use of labels. This is often used to define low level interrupts. See the chapter on Interrupts, for more details on writing low level interrupts.

Creating defining words in assembler

The cross compiler allows you to define the run-time (**DOES>**) part of a defining word in assembler. To do this use **;CODE** in the form:

```
: <name>
  CREATE
   . .
  ;CODE
    . .
END-CODE
```

An example is shown in figure 8.

```
: VARIABLE  \ — ; — addr [child]
  CREATE                                    \ create a child header
    HERE 2+ , 0 ,                           \ lay address of data
  ;CODE                                     \ run time in assembler
    hl inc  hl inc  hl inc          \ skip CFA
    e, (HL) ld
    hl inc d, (hl) ld               \ get address in RAM
    de push  next,                          \ place on stack
END-CODE
```

□□□□□□□ 8 □ □□□ □□ ;□□□□

Structured programming

Three facilities are available to give you the advantages of structured programming, in assembler:

- control structures
- labels
- local labels

Control structures

There are assembler equivalents to the Forth control structures. The available structures are:

cc IF, ... THEN,
cc IF, ... ELSE, ... THEN,
BEGIN, ... cc UNTIL,
BEGIN, ... CC WHILE, ... REPEAT
BEGIN, ... AGAIN,

where cc is one of the condition codes in table 4

Mnemonic	Condition	Mnemonic	Condition
Z,	zero	P,	positive or 0
NZ,	non-zero	M,	negative
CY,	carry set	PE,	even parity
NCY,	carry not set	PO,	odd parity

Table 4 - Available condition codes

Labels

Labels can be used to mark a place in assembler code. That place can then be referenced in other areas of code.

Creating a label

Labels can be defined by using the command **L:**. It is used in the form:

l: <name>

where <name> is the name you want to call the label.

Referencing a label

A label is referenced by stating its name. For example,

```
Z, <name> JP
```

will assemble to 'jump if zero' to <name>.

Local labels

If you need to use labels within a code definition, you may use the local labels provided. These are used just as normal labels in the assembler, but some restrictions apply:

- there is a maximum of ten labels
- the names are in the form L$n where n is in the range 1 to 10
- a reference is valid until the next occurence of **CODE** or **;CODE**

Creating a local label

To define a local label use L$n:, where n is a number from one to ten. For example:

```
L$1:
```

Referencing a local label

To reference a local label, type its name. For example,

```
L$1 JP
```

assembles code for a jump to L$1.

Creating macros

A macro is a word that lays down code 'in-line' within an assembler definition. They are normally used when there is a repetitive use of a series of opcodes.

Defining a macro

A macro is defined using colon and semi-colon. It must also be defined in the cross compiler's vocabulary, **ASM-ACCESS**. The place to create a macro is in the control file and it **must** be defined before the word **CROSS-COMPILE**. As an example, a macro **NEXT,** is shown in figure 9. This is defined as a macro, so each time it is used, its code is layed down. This makes it quicker than calling a subroutine.

```
\ switch to the cross-compilers ASM-ACCESS vocabulary
ONLY FORTH ALSO C-C ALSO ASSEMBLER
ALSO ASM-ACCESS DEFINITION

: NEXT,                          \ — ; lay inline code
  A, (BC) LD   BC INC  L, A LD
  A, (BC) LD   BC INC  H, A LD
  (HL) JP
;

\ switch back to forth vocabulary
ONLY FORTH DEFINITIONS
```

□□□□□□□ 9 □ □□□□□□□□ □□□□□ □□□□□□□□□□□

Using a macro

A macro is used by stating its name. For example, in a **CODE** definition, **NEXT,** would be a macro.

Addressing modes

This section lists the syntax used for the addressing modes. It is as close as possible to the usual Hitachi syntax, but the operands have to be separated by spaces. For example, where you would normally have written

LD (HL),A

in the Forth assembler you have to write:

(HL), A LD

If no other indications are given, the default addressing mode (the one chosen if you do not indicate which one is required) is extended, where valid. The usual 64180 convention of operand order of destination followed by source is adhered to. The Forth assembler accepts the following addressing mode formats.

Extended Addressing

Extended addressing is the default addressing mode of the 64180 assembler. No indicator is required to specify the address:

A, 055AA LD

moves the contents of address 055AA into accumulator register A. Address arithmetic may also be performed.

Immediate Addressing

Immediate addressing requires the # indicator to specify the immediate number to use:

A, # 0AA LD

moves the value 0AA into accumulator register A.

Indexed Addressing

Indexed addressing is specified by putting the index register in parentheses and preceeding it with the displacement. The displacement must always be present, even if it is 0. It cannot be forward referenced and must lie in the range -80h..7Fh because it is added to the index register as a signed displacement.:

```
A, 5 (IX) LD
0 (IY) DEC
```

Address arithmetic may be performed on the offset.

Implied Addressing

On zero operand instructions the register to use is implied in the instruction itself, but for instructions which require an operand you must specify the target register:

```
CPL
A, # 55 AND
(C), 2 TSTIO
```

Register Direct Addressing

The register to use is specified as one of the operands in the instruction:

```
A, B LD
D INC
```

Register Indirect Addressing

The register to use is specified as one of the operands in the instruction and is in parentheses:

```
(BC), A LD
(HL) RL
```

Relative Addressing

This is only used by the branch instructions. The branch displacement (relative to the PC) is contained in the instruction:

loopstart DJNZ
Z, 55 JR

I/O Addressing

This is only used by the I/O instructions.

33 A OUT0

Number Bases

The number base in the Forth assembler can be indicated by the words BINARY DECIMAL and HEX. In addition, numbers prefixed by the '$' '#' and '%' characters are treated as special cases. These characters affect the number base for that number only. Note that the characters '$' and '%' follow Motorola usage. Note that the '#' symbol attached to a number is not the same as the # word that indicates immediate addressing.

Symbol	Base	Example
$	hex	$55AA
#	decimal	#1234
%	binary	$1011001

Instruction syntax

The instructions are shown alphabetically with all their addressing forms. Note this section describes the 64180 instruction set, so not all instructions will be available for use on a Z80.

Notation

The notation follows the Hitachi conventions and is shown in table 5.

Text	Meaning	Text	Meaning
g or g'	Registers A B C D E H L	v	Restart Address
ww	Registers BC DE HL SP	m or n	8 bit data
xx	Registers BC DE IX SP	mn	16 bit data
yy	Registers BC DE IY SP	r	8 bit register
zz	Registers BC DE HL AF	R	16 bit register
b	Bit number (0-7)	d or j	8 bit signed displacement
f	Flag		

Table 5 - 64180 cross assembler notation

Instruction list

Conventional	Forth
ADC A, g	A, g ADC
ADC A, (HL)	A, (HL) ADC
ADC A, (IX+d)	A, d (IX) ADC
ADC A, (IY+d)	A, d (IY) ADC
ADC A, m	A, # m ADC
ADC HL, ww	HL, ww ADC
ADD A, g	A, g ADD
ADD A, (HL)	A, (HL) ADD
ADD A, (IX+d)	A, d (IX) ADD
ADD A, (IY+d)	A, d (IY) ADD
ADD A, m	A, # m ADD
ADD HL, ww	HL, ww ADD
ADD IX, xx	IX, xx ADD
ADD IY, yy	IY, yy ADD
AND g	A, g AND
AND (HL)	A, (HL) AND

AND (IX+d)	A, d (IX) AND	
AND (IY+d)	A, d (IY) AND	
AND m		A, # m AND
BIT b, g		b g BIT
BIT b, (HL)	b (HL) BIT	
BIT b, (IX+d)	b d (IX) BIT	
BIT b, (IY+d)	b d (IY) BIT	
CALL f, mn	f, mn CALL	
CALL mn		mn CALL
CCF		CCF
CPD		CPD
CPDR		CPDR
CP g		A, g CP
CP (HL)		A, (HL) CP
CP (IX+d)		A, d (IX) CP
CP (IY+d)		A, d (IY) CP
CP m		A, # m CP
CPI		CPI
CPIR		CPIR
CPL		CPL
DAA		DAA
DEC g		g DEC
DEC (HL)		(HL) DEC
DEC IX		IX DEC
DEC IY		IY DEC
DEC (IX+d)	d (IX) DEC	
DEC (IY+d)	d (IY) DEC	
DEC ww		ww DEC
DI		DI
DJNZ j		j DJNZ
EI		EI
EX AF, AF’	AF, AF’ EX	
EX DE, HL		DE, HL EX
EX (SP), HL	(SP), HL EX	
EX (SP), IX	(SP), IX EX	
EX (SP), IY	(SP), IY EX	
EXX		EXX
HALT		HALT

IM 0		0 IM
IM 1		1 IM
IM 2		2 IM
IN A, (m)		A, m IN
IN g, (C)		g, (C) IN
IN0 g, (m)	g, m IN0	
INC g		g INC
INC (HL)		(HL) INC
INC IX		IX INC
INC iY		IY INC
INC (IX+d)	d (IX) INC	
INC (IY+d)	d (IY) INC	
INC ww		ww INC
IND		IND
INDR		INDR
INI		INI
INIR		INIR
JP f, mn		f, mn JP
JP (HL)		(HL) JP
JP (IX)		(IX) JP
JP (IY)		(IY) JP
JP mn		mn JP
JR j		j JR
JR C, j		C, j JR
JR NC, j		NC, j JR
JR Z, j		Z, j JR
JR NZ, j		NZ, j JR
LD A, (BC)	A, (BC) LD	
LD A, (DE)	A, (DE) LD	
LD A, I		A, I LD
LD A, (mn)	A, mn LD	
LD A, R		A, R LD
LD (BC), A	(BC), A LD	
LD (DE), A	(DE), A LD	
LD g, g’		g, g’ LD
LD g, (HL)	g, (HL) LD	
LD g, (IX+d)	g, d (IX) LD	
LD g, (IY+d)	g, d (IY) LD	
LD g, m		g, # m LD
LD (HL), g	(HL), g LD	
LD (HL), m	(HL), # m LD	
LD HL, (mn)	HL, mn LD	
LD I, A		I, A LD

LD IX, mn		IX, # mn LD
LD IY, mn		IY, # mn LD
LD IX, (mn)	IX, mn LD	
LD IY, (mn)	IY, mn LD	
LD (IX+d), g	d (IX), g LD	
LD (IY+d), g	d (IY), g LD	
LD (IX+d), m	d (IX), # m LD	
LD (IY+d), m	d (IY), # m LD	
LD (mn), A	mn A LD	
LD (mn), HL	mn HL LD	
LD (mn), IX	mn IX LD	
LD (mn), IY	mn IY LD	
LD (mn), ww	mn ww LD	
LD R, A		R, A LD
LD SP, HL		SP, HL LD
LD SP, IX		SP, IX LD
LD SP, IY		SP, IY LD
LD ww, mn		ww, # mn LD
LD ww, (mn)	ww, mn LD	
LDD		LDD
LDDR		LDDR
LDI		LDI
LDIR		LDIR
MLT ww		ww MLT
NEG		NEG
NOP		NOP
OR g		A, g OR
OR (HL)		A, (HL) OR
OR (IX+d)		A, d (IX) OR
OR (IY+d)		A, d (IY) OR
OR m		A, # m OR
OTDM		OTDM
OTDMR		OTDMR
OTDR		OTDR
OTIM		OTIM
OTIMR		OTIMR
OTIR		OTIR
OUT (m), A	m A OUT	
OUT (C), g	(C), g OUT	
OUT0 (m), g	m g OUT0	

OUTD		OUTD
OUTI		OUTI
POP IX		IX POP
POP IY		IY POP
POP zz		zz POP
PUSH IX		IX PUSH
PUSH IY		IY PUSH
PUSH zz		zz PUSH
RES b, g		b g RES
RES b, (HL)	b (HL) RES	
RES b, (IX+d)	b d (IX) RES	
RES b, (IY+d)	b d (IY) RES	
RET		RET
RET f		f, RET
RETI		RETI
RETN		RETN
RL g		g RL
RL (HL)		(HL) RL
RL (IX+d)		d (IX) RL
RL (IY+d)		d (IY) RL
RLA		RLA
RLC g		g RLC
RLC (HL)		(HL) RLC
RLC (IX+d)	d (IX) RLC	
RLC (IY+d)	d (IY) RLC	
RLCA		RLCA
RLD		RLD
RR g		g RR
RR (HL)		(HL) RR
RR (IX+d)		d (IX) RR
RR (IY+d)		d (IY) RR
RRA		RRA
RRC g		g RRC
RRC (HL)		(HL) RRC
RRC (IX+d)	d (IX) RRC	
RRC (IY+d)	d (IY) RRC	
RRCA		RRCA
RRD		RRD
RST v		v RST

SBC A, g		A, g SBC
SBC A, (HL	A, (HL) SBC	
SBC A, (IX+d)		A, d (IX) SBC
SBC A, (IY+d)		A, d (IY) SBC
SBC A, m		A, # m SBC
SBC HL, ww	HL, ww SBC	
SCF		SCF
SET b, g		b g SET
SET b, (HL)	b (HL) SET	
SET b, (IX+d)	b d (IX) SET	
SET b, (IY+d)	b d (IY) SET	
SLA g		g SLA
SLA (HL)		(HL) SLA
SLA (IX+d)	d (IX) SLA	
SLA (IY+d)	d (IY) SLA	
SLP		SLP
SRA g		g SRA
SRA (HL)		(HL) SRA
SRA (IX+d)	d (IX) SRA	
SRA (IY+d)	d (IY) SRA	
SRL g		g SRL
SRL (HL)		(HL) SRL
SRL (IX+d)	d (IX) SRL	
SRL (IY+d)	d (IY) SRL	
SUB g		A, g SUB
SUB (HL)		A, (HL) SUB
SUB (IX+d)	A, d (IX) SUB	
SUB (IY+d)	A, d (IY) SUB	
SUB m		A, # m SUB
TST g		A, g TST
TST m		A, # m TST
TST (HL)		A, (HL) TST
TSTIO m		(C), m TSTIO
XOR g		A, g XOR
XOR (HL)		A, (HL) XOR
XOR (IX+d)	A, d (IX) XOR	
XOR (IY+d)	A, d (IY) XOR	
XOR m		A, # m XOR

Glossary

This glossary details the words provided within the cross-assembler to control the use of the assembler.

;CODE — I
"semi-code"

Used in the form:

: <namex> CREATE .... ;CODE ... END-CODE

Stops compilation, and enables the assembler. This word is used with **CREATE** to produce defining words whose run-time portion is written in code, in the same way that **CREATE** ... **DOES>** is used to create high level defining words.
The data structure is defined between **CREATE** and **;CODE** and the run-time action is defined between **;CODE** and **END-CODE**. The current value of the data stack pointer is saved by **;CODE** for later use by **END-CODE** for error checking. When <namex> executes the address of the data area will be found on the processor stack, from which it must be removed.
A version of **VARIABLE** for use in a ROM-based system might be:

```
: VARIABLE                    \ — ; — addr [child] ; ROMable
  CREATE  HERE 2+ ,  0 , \ lay pointer, and 0
  ;CODE                                \ start of code portion
   hl inc hl inc hl inc  \ Skip CFA
   e, (hl) ld
   hl inc d, (hl) ld     \ get addr in RAM
   de push    NEXT,      \ return it and exit
END-CODE

VARIABLE TEST-VAR
```

ASSEMBLER —
"assembler"

Starts a section of assembler code and turns on the assembler, but without generating a dictionary header. This action is particularly useful for generating the start-up code. Examples of this can be found in CODE180.FTH.

CODE —
"code"

A defining word used in the form:

CODE <name> ... END-CODE

Creates a dictionary entry for <name> to be defined by a following sequence of assembly language words. Words thus defined are called code definitions. **CODE** stores the current data stack pointer for later error checking by **END-CODE**.

END-CODE —

"end-code"

Terminates a code definition and checks the data stack pointer against the value stored when **;CODE** or **CODE** is executed. The assembler is disabled. See: **CODE ;CODE**

FORTH —

"forth"

Terminates a section of assembler code started by the word **ASSEMBLER** and turns off the assembler.

IS-ACTION-OF addr —

"is action of"

Used to tell the cross compiler that the given address is to be used as the run time action of the word whose name follows. Usually found in code definitions, but can also be used for high level definitions. For example:

```
ASSEMBLER
HERE IS-ACTION-OF CONSTANT
.....
FORTH
```

```
ASSEMBLER
HERE IS-ACTION-OF <high-level-definer>
JSR DODOES
FORTH
] ..... EXIT [
```

Blank Page

Multitasker

The multitasker supplied with the MPE development system can greatly simplify complex tasks by breaking them down into manageable chunks. This chapter leads you through:

- initialising the multitasker
- writing a task
- communicating between tasks
- handling events

The multitasker is in the file MULTI180.FTH in the \ROM directory.

Note: The multitasker cannot be used with Umbilical Forth

Initialising the multitasker

The multitasker needs to be initialised before use. At compile time the cross compiler must be told the total number of tasks that your system requires and at run-time, all the tasks must be initialised.

Setting the number of tasks

The number of tasks is set in your control file. It is in the form:

```
xxxx EQU #TASKS
```

where xxxx is 8, by default, but can be set to a different number. This reduces the amount of memory that is allocated to all the tasks, so leaving more RAM for your application.

Starting the multitasker

To start the multitasker, use **MULTI**. **MULTI** starts the scheduler so new tasks can be added.

Stopping the multitasker

To stop the multitasker, use **SINGLE**.

Writing a task

Tasks are very straightforward to write, but the way tasks are scheduled needs to be understood.

Using the scheduler

The multitasker is software scheduled. This means that each task relinquishes control back to the scheduler when it's ready. This is different from a pre-emptive scheduler where the scheduler interrupts a task. Two words are supplied so that a task can relinquish control back to the scheduler, **PAUSE** and **WAIT**.

Using PAUSE

The word **PAUSE** passes control back to the scheduler which executes all the other tasks once, then returns back to this task.

Using WAIT

The word **WAIT** suspends a task for a certain number of schedules. It is used in the form:

n WAIT

where n is the number of schedules to suspend the task. When **WAIT** is used, it transfers control to the scheduler. The scheduler does not execute this task again until all the other tasks have been executed n times.

```
: TASK1                 \ — ; An example task
  BEGIN                                 \ Start an endless loop
    7 EMIT              \ Produce a beep
    1000 WAIT           \ Reshedule 1000 times
  AGAIN                                 \ Go round again
;
```

□□□□□□ 10 □ □□□□□□□□□□□□ □□□□□□□

An example

An example task is shown in figure 10. The task is an endless loop with the word **WAIT** embedded in it. When the word **WAIT** is executed, the scheduler reschedules to the next task. The scheduler will not run this task until it has run all other tasks 1000 times. Each time the task is executed, it will emit a beep.

Task dependant variables

An area of memory is set aside for each task. This memory contains user variables which contain task specific data. For example, the current base is normally a user variable as it can vary from task to task.

Defining a user variable

A user variable is defined in the form:

n USER <name>

where n is the nth byte in the user area.

Using a user variable

A user variable is used in the same way as a normal variable. By stating its name, its address is placed on the stack, which can then be fetched using **@** and stored by **!**.

Initialising a task

A task needs to be initialised before it is run. To to this it needs to be assigned to a task number. The task number can range from zero to the maximum number of tasks stated in the control file. A task is assigned in the form:

ASSIGN TASK1 N TO-TASK

where **TASK1** is your task word and n is the task number. For example, to initialise the task in figure 10, to task 1, you type:

ASSIGN TASK1 1 TO-TASK

The task number is used to control the task.

Controlling tasks

Tasks can be controlled in the following ways:

- activated
- suspended for a number of schedules
- halted
- restarted after its been halted

You can also stop the current task.

Starting a task

A task can be started by activating it. To activate a task, use

n ACTIVATE

where n is the task number.

Stopping a task

A task may be stopped for a number of cycles of the scheduler or temporarily suspended. A task may also stop itself.

Stopping for a number of cycles

To stop the current task for a number of cycles, use **WAIT. WAIT** is used in the form,

n WAIT

where n is the number of schedules to stop the task.

Temporarily stopping a task

To temporarily stop a task, use **HALT**. **HALT** is used in the form,

n HALT

where n is the task to be stopped.

To restart a stopped task, use **RESTART**. **RESTART** is used in the form,

n RESTART

where n is the task to restart.

Stopping the current task

To stop the current task (i.e. stop itself) use **STOP.** This word should obviously be used with caution. **STOP** is used in the form,

STOP

Handling messages

An essential feature of the multitasker is the ability to send and receive messages between tasks.

Sending a message

To send a message to another task, use the word **SEND-MESSAGE**. **SEND-MESSAGE** is used in the form:

message task# SEND-MESSAGE

where message is a 16-bit message and task# is the number of the task to send the message to. The message can be data, an address or any other type of information but its meaning must be known to the receiving task.

Receiving a message

To receive a message, use **GET-MESSAGE**. **GET-MESSAGE** suspends the task until a message arrives. When a message is received the task is re-activated and the sending task number and the data is returned.

Creating events

Events are analogous to interrupts. Whereas interrupts happen on hardware signals, events happen under software control.

Writing an event

An event is a normal Forth word. An event is associated to a task so that when the event is triggered, the task is activated. Therefore, an event is usually used as initialisation for a task.

Initialising an event

Events are initialised in a similiar way to tasks. They are assigned in the form,

ASSIGN EVENT1 n TO-EVENT

where EVENT1 is your event handler and n is the task number of the task that it is to be associated with.

Triggering an event

There are two ways of triggering an event:

- using **SET-EVENT**
- setting a bit in the status word

Using Set-event

SET-EVENT is a word which sets an event flag for a task. Once the event flag is set, the tasker will execute the event before it switches to the task. The task is also activated.

Bit	when set	when reset
7	Task is running	Task is halted
6	Message pending	No messages
5	Event has been triggered	No events

Table 6 - A task's status word

Field	Contains	Size
TCBSP	Data stack pointer	word
TCBST	Task status byte	byte
TCBID	Task number of message sender	byte
TCBMSG	Message code or address	word
TCBEVENT	CFA of word run by task’s event handler	word
TCBAC-TION	CFA of main task word	word

Table 7 - Multitasker data structure

Setting a bit in the status word.

A bit can be set in a task’s status word which indicates to the multitasker that an event has taken place. This method can be used to trigger an event from a hardware interrupt. Refer to ‘The multitasker’s internals’ later in the chapter for details on the status byte.

Clearing an event

To stop an event handler being run, use **CLEAR-EVENT**.

The multitasker’s internals

A multitasker tries to simulate many processors with just one processor. It works by rapidly switching between each task. On each task switch it saves

the current state of the processor, and restores the state that the next task needs.

The Forth multitasker is software scheduled. This means that each task relinquishes control to the scheduler, which then switches to the next task. In this way less processor state information needs to be saved.

The scheduler's data structure

The Forth multitasker creates a task control block for each task. The task control block (TCB) is a data structure which contains information relevant to a task (figure). The status byte (TCBST) contains information on the execution of the task and its event (figure).

A simple example

The following example is a simple demonstration of the multitasker. Its simple role is to display a hash (#) every so often, but leaving the forground Forth running. To use the multitasker you must cross-compile the file MULTI180.FTH into your target.

Defining a simple task

The following code defines a simple task called TASK1. It displays a # every 1000 schedules.

```
VARIABLE DELAY                  \ time delay between #'s
 1000 DELAY !                   \ initialise time delay

: TASK1                 \ — ; task to display #'s
 ASCII $ EMIT                   \ Display a dollar ($)
 BEGIN                          \ Start continuous loop
  ASCII # EMIT                  \ Display a hash (#)
  DELAY @ WAIT                  \ Reschedule Delay times
 AGAIN                          \ Back to the start ...
;
```

Initialising the multitasker

Before any tasks can be activated, the multitasker must be initialised. This is done with the following code:

```
INIT-MULTI
MULTI
```

The word **INIT-MULTI** initialises all the multitasker's data structures and **MULTI** switches to multitasking. These words need only be executed once in a multitasking system.

Assigning the example task to a task number

In a multitasking system, tasks are represented by numbers. Therefore, each task must be assigned to a task number. For this example you type:

```
ASSIGN TASK1 1 TO-TASK
```

This assigns the word **TASK1** to task number 1. It can be assigned to any task upto the number of tasks defined in the system (defined by **#TASKS** in the control file).

Activating the example task

To activate (run) the example task, type:

```
1 ACTIVATE
```

This will activate task number one. Immediately you will see a dollar and a hash ($#) displayed. If you press <return> a few times, you notice that the Forth is still running. After a few seconds another hash will appear. This is the example task working in the background.

Controlling the example task

The example task can be controlled in several ways:

- the rate of generation of hashes can be changed
- it can be halted
- once halted it can be restarted
- it can be started from scratch

Changing the rate of hashes

The rate of production of hashes can be changed by changing the variable **DELAY**. Try:

2000 DELAY !

This changes the number of schedules that the example tasks makes between displaying hashes to 2000. Therefore the rate of displaying hashes halves.

Halting the example task

The task is halted by typing the tasks number followed by **HALT**:

1 HALT

You notice that the hashes are not displayed.

Restarting the halted task

The task is restarted by the word **RESTART**. Type the task number followed by **RESTART**:

1 RESTART

You notice that the hashes are displayed again.

Restarting the task from scratch

To restart the task from scratch, just activate it again:

1 ACTIVATE

You notice the dollar and the hash ($#) are displayed, followed by hashes (#).

Glossary

This glossary contains details of the major words in the multi-tasking system. Other words exist, but are only used as fractions of the words below.

?EVENT —
"query-event"

If the current task's event flag is set, the flag is reset and the event handler is executed.

ACTIVATE task# —
"activate"

Initialises and starts the given task number. Task 0 is Forth itself and was activated when Forth started. Note that **ACTIVATE** causes the task to start from the very beginning. If the task was halted, and execution should resume where it left off, use **RESTART** instead.

CLR-EVENT-RUN —
"clear-event-run"

Clears the event run flag for the current task. This is bit 4 in the task status byte.

EVENT? — t/f
"event-query"

Returns true if the event triggered bit has been set in the current task's status byte.

GET-MESSAGE — message task#
"get-message"

Waits for a message and returns the message and the sending task.

HALT task# —
"halt"

Halts the task whose number is given. Do not halt task 0. Halting a task prevents it responding to messages or events.

INIT-MULTI —
"init-multi"

Initialises the multi-tasker, task 0, and starts the multi-tasker. Just include this word in **COLD** to kick the multi-tasker into action.

INIT-TCBS —
"init-t-c-bees"

The main part of the multi-tasker reset process.

MSG? task# — t/f
"message-query"

Returns true if the task is holding a message, and is therefore not free to receive another one.

MULTI —
"multi"

Turns the multi-tasker on, by clearing the bit in the TASK# byte in internal RAM that inhibits the scheduler.

PAUSE —
"pause"

Waits for one iteration of the scheduler. Equivalent to:

```
1 WAIT
```

RESTART task# —
"restart"

Restarts a task that was halted by **HALT** or **WAIT**. Unlike **ACTIVATE**, the task resumes where it left off.

SELF — task#
"self"

Returns the task number of the current task. Useful with **MSG?** in particular to determine whether or not a message has been received by the task.

SEND-MESSAGE message task# —
"send-message"

Sends a message to the given task. The message address can be used on its own, or as a pointer to an extended message.

SINGLE —
"single"

Turns off the multi-tasker by setting the scheduler disable bit in the **TASK#** byte in internal RAM.

STATUS — n
"status"

Returns the task status byte of the current task but with the top bit (bit 7) masked off. If this value is non-zero, the task has been awakened for a reason other than for normal running.

TCBS — addr
"t-c-b-st"

A label, NOT a word, that returns the start address in DATA RAM of the table holding the action words for all the tasks. In some systems this is implemented as a constant for visibility.

TO-EVENT cfa task# —
"to-event"

Sets the CFA of a Forth word as the action to run when the task's event trigger is set.

ASSIGN <word> <n> TO-EVENT

TO-TASK cfa task# —
"to-task"

Stores the CFA of the word forming the task action in the task table entry for the task.

ASSIGN <word> <n> TO-TASK

WAIT n —
"wait"

Suspends the current task for n iterations of the scheduler. If n is 0, the task is suspended until a message or event is received.

WAIT-EVENT/MSG —
"wait-event-or-message"

The current task is suspended until it receives a message or an event trigger. The words **MSG?** and **EVENT?** can be used to determine whether a message or an event trigger terminated the wait. Note that if an event trigger is received, the event handler will have been called, and the event run flag (bit 4 in the status byte) will be set.

Interrupts

This chapter describes how to write interrupt handlers in both Forth and assembler. It details how to set up and control interrupt handlers.

The 64180 interrupt mechanism

When an interrupt occurs and is accepted, the action a 64180 processor takes is dependent on the interrupt mode that has been set by the user. This can be done by executing an IM instruction in assember. In mode 2, the processor branches through a location (a 'vector') in memory. The code at the address is then executed. The vector table typically starts at address 0000h in memory, although this is user selectable by writing into the I register. The address of the vector used is dependent on the source of the interrupt, and this is also user selectable. The address of the internal interrupts on a 64180 can be selected by writing into the top three bits of the IL register. For more information on the interrupts for the processor, refer to the processor's user guide.

Writing Forth interrupt handlers

A Forth interrupt service routine (ISR) is just like any other Forth word. It can therefore be tested and debugged like a normal Forth word. Only when the word is fully tested need it be assigned to an interrupt.

Interrupts using Mode 2

This section describes how to write a high-level interrupt handler for the 64180. Below is a formula, which you must copy for each interrupt you wish to process with a high-level Forth word. It is taken from the file INTERUPT.FTH in the X180\ROM directory, and is an example for the inter-

nal timer 0 interrupt. This uses the same mechanism as mode 2 interrupts on the Z80

First, write the word which is to be the action of the interrupt, and test it as far as possible:

```
VARIABLE COUNTER                    \ to count ticks

: EXAMPLE-ISR           \ — ; interrupt handler
  RESET-TIMER                       \ clear int source
  1 COUNTER +!                      \ inc counter
;
```

Now define the following piece of assembler:

```
DEFER TIMER0-ISR

HEX
L: TIMER0-INT
 ] TIMER0-ISR RETI [
ASSEMBLER                           \ this code uses 8 bytes so
L: TIMER0-INT0                      \ can be used in RST xx vector
  AF PUSH    BC PUSH      \ save status and IP
  BC, # TIMER0-INT LD     \ point to Forth and run
  COMMON-ISR JP                     \ join common code
FORTH
```

The first label, TIMER0-INT in this case makes a space to contain the name of the word to execute for the interrupt. It also contains the name of the word which returns from a high-level interrupt. Between the square brackets are the name of the word to execute, TIMER0-ISR in this case, and the name of the word RETI whose job is to make a return from interrupt.

The second label, TIMER0-INT0 in this case, is the interrupt routine which 'kicks off' the high-level Forth routine. You must copy this into your routine. Note that TIMER0-INT is referred to by TIMER0-INT0. The label COMMON-ISR, and the word RETI are both defined in the file INTERUPT.FTH which is part of the kernel system. These words and labels define the interrupt routine but the vector must still be set. This is performed with the line

```
TIMER0-INT0 INT-PAGE INT-PAGE-L + 4 + !
```

This sets the address of TIMER0-INT0 as the vector for the interrupt. On a 64180 the TIMER 0 vector occupies bytes 4 & 5 of the block of memory selected by the values contained in the I and IL registers. These values are available as INT-PAGE and INT-PAGE-L respectively. You should consult the reference books for the processor for more details or more vectors in specific variants of the processor.

Since the TIMER0-ISR word is deferred in the example above, you must assign its action in the word which starts your application running. In the example above, the following code performs this task:

```
: RUN-EXAMPLE              \ —
  ASSIGN EXAMPLE-ISR TO-DO TIMER0-ISR
  START-TIMER
;
```

You should note that it would be quite possible to patch EXAMPLE-ISR into the TIMER0-INT word directly, instead of going via a deferred word. This would save a small amount of memory at the expense of not being able to change the interrupt action on the fly.

The START-TIMER, RESET-TIMER and STOP-TIMER words simply enable the interrupts and clear the source by reading and writing the TCR and TMDR0 registers as appropriate. When the target runs, the interrupt will be enabled, and then the timer overflow example will increment the variable. This can be seen by examining the value of the variable from time to time.

Interrupts under Modes 0 and 1

The interrupt handler mechanism described previously assume the use of mode 2 interrupts. However similar mechanisms can be used for mode 0 and mode 1 interrupts, the only difference being the way that the vector is patched. Instead of generating the vector address from a number of equates, you should use the ORG instruction to force assembly at the correct location. For example

```
HEX
HERE                                    \ preserve current location
038 ORG                                 \ force assembly at location 38h

ASSEMBLER
L: RST38-ENTRY
   MY-INT JP                \ Jump to handler
FORTH

ORG                                     \ restore location
```

would generate code for calling a routine called MY-INT using an RST 38 instruction. This would therefore be suitable for use with mode 0 and mode 1 interrupts. Note that the TIMER0-INT0 code presented earlier in the chapter is 8 bytes long. This means that this code can be be put into a mode 0 vector slot directly, without having to make a separate jump.

Some common problems

There are a few common problems that might cause an interrupt not to work correctly:

- a stack fault
- the source is not cleared
- the interrupts are not enabled

Stack fault

An interrupt service routine can use the stack while it is executing, but must clear up the stack before returning from the interrupt. The normal symptom of a stack fault is that the interrupt handler runs but then the target board crashes, either immediately or after a length of time.

Source is not cleared

Once an interrupt handler is triggered by an interrupt, the source of the interrupt must be told that the interrupt is being serviced. If this is not done, the source of the interrupt will carry on generating interrupts. Normally this appears as the interrupt handler executing once and then the target board 'locking'.

Interrupts are not enabled

Interrupts need to be enabled with **EI** before any interrupts will be serviced. The vectors must be set up before the interrupts are enabled.

Writing assembler interrupt handlers

Writing an interrupt service routine in assembler is straightforward. The code can be written in the form shown below, which is taken from the serial line driver in DARTCTCI.FTH in the ROM\DRIVERS subdirectory. The entry point is indicated by the label RX-INT. For more information on how to write assembler definitions see chapter 6.

```
ASSEMBLER
L: RX-INT
  AF PUSH
  A, B-DATA IN  RX-CHAR A LD              \ read char and stash
```

```
  A, # -1 LD    RX-AVAIL A LD                \ flag char as read
  AF POP   EI   RETI             \ all done
FORTH
```

Setting the interrupt

The interrupt code above can be set to a vector by entering in your source code,

```
INT-PAGE 010 +  EQU  DART-VECTOR

RX-INT  DART-VECTOR 4 +  !                   \ set serial interrupt vector
```

The DART-VECTOR equate is also used when initialising the UART so that it knows which addresses have been allocated to it. You will see code similar to this in the INTERUPT.FTH and DRIVERS\DARTCTCI.FTH files. The same formula is used for every interrupt in the system.

Controlling the interrupts

Interrupts can be in one of two states, enabled or disabled.

Enabling interrupts

To enable interrupts use **EI**. Once **EI** has been executed, all interrupts are enabled.

Disabling interrupts

To disable interrupts use **DI**. Once **DI** has been executed, all interrupts are disabled.

Interrupt Handlers in detail

A conventional interrupt handler written in assembler simply saves any registers and fixed locations it will use, then calls or executes the required routine, and then restores the previous state of registers and locations, and

returns. The Forth interrupt handlers work in the same way, with a few additions.

Interrupt handlers written entirely in assembler are written in the usual way. When a Forth **VARIABLE** is referred to by name inside a code section, its data address in RAM is returned. Thus variables can be shared between high level routines and subroutines or assembler interrupt handlers.

The code for the interrupt handler can be found in the file INTERUPT.FTH in the ROM subdirectory. This source code can be modified and updated as required. The interrupt handler code can be adapted for single-chip use.

Interrupt Structure

A separate user area is reserved for the interrupt handler, and is shared by all the high-level interrupt handlers. Consequently high-level Forth interrupts cannot be nested if user variables are changed, although an interrupt that does not use the Forth interrupt handler can be nested.

When an interrupt that has a high-level handler occurs, the processor status is pushed onto the stack. This includes the Interpretive Pointer and User Pointer. These pointers are then loaded with the required values for interrupt processing.

The Interpretive Pointer is set by the interrupt code to point to an area of memory that contains the CFAs of two words. The first is the CFA of a word to run the interrupt. The second is the CFA of the return from interrupt word.

```
l: some-interrupt
  ] action reti  [
```

Setting an interrupt

The return from interrupt is provided by the second word of the table entry. This means that the action word need contain no return action, and consequently it may be tested from the keyboard. The file ROM\INTERUPT.FTH contains an example interrupt using one of the timers on the 64180. Since mode 2 interrupts are vectored, the final requirement is to set the vector itself.

Note that interrupt handlers written in assembler do not need to use these structures at all. They must still, however, set the vector to use as appropriate.

Interrupt Protection

The word **SAVE-INT** saves the current interrupt status, and disables interrupts, which can then be restored by **RESTORE-INT** which restores the interrupt mask to the state saved by **SAVE-INT**. If simple enabling and disabling of interrupts is required, the words **EI** and **DI** respectively perform these functions. **SAVE-INT** and **RESTORE-INT** are particularly necessary for critical sections of code, and for implementing semaphores.

Glossary

This glossary contains details of the major words in the interrupt system. Other words exist, but are only used as fractions of the words below. The source code for all these words may be found in INTERUPT.FTH.

DI —
"d-i"

Disables interrupts.

EI —
"e-i"

Enables interrupts.

RESTORE-INT n —
"restore-int"

Restore the interrupt enable state previously saved by **SAVE-INT**.

SAVE-INT — n
"save-int"

Saves the current state of the interrupt enable on the stack, and disables interrupts. See **RESTORE-INT**.

Software floating point

Although most applications only require integer arithmetic, some do require floating point. Therefore software floating point is supplied with the cross-compiler and the target Forth.

The cross-compiler has a more limited floating point support than the target, this means that some words are avaliable within colon definitions, but not outside them.

Entering floating point numbers

Floating point numbers can be entered in two forms, 1.234 and 0.1234e1

Floating point numbers are compiled as literal numbers when in a colon definition and placed on the cross-compiler's stack when outside a definition.

The form of floating point numbers

A floating point number is placed on the Forth stack. It consists of three 16-bit numbers. Two for the mantissa and one for the exponent. The mantissa is normalised.

Creating variables

To create a variable, use **FVARIABLE**. **FVARIABLE** works in the same way as **VARIABLE**. For example, to create a floating point variable called VAR1 you code:

```
FVARIABLE VAR1
```

When VAR1 is used, it returns the address of the floating point number.

Accessing variables

Two words are used to access floating point variables, **F@** and **F!**. These are analogous to **@** and **!**.

Creating constants

To create a floating point constant, use **FCONSTANT**. **FCONSTANT** is analogous to **CONSTANT.** For example, to generate a floating point constant called **CON1** with a value of 1.234, you enter:

```
1.234 FCONSTANT CON1
```

When the **CON1** is executed, it returns 1.234 on the Forth stack.

Using the supplied words

The supplied words split into several groups:

- sines, cosines and tangents
- arc sines, cosines and tangents
- arithmetic functions
- logarithms
- powers
- displaying floating point numbers
- inputting floating point numbers

The following functions only exist as target words so you cannot use them in calculations in your source code when outside a colon definition.

Calculating sines, cosines and tangents

To calculate a sine, cosine and tangent, use **FSIN**, **FCOS** and **FTAN** respectively. They take an angle in either degrees or radians, depending on which is set at the moment. See Setting degrees or radians.

Calculating arc sines, cosines and tangents.

To calculate the arc sine, cosine and tangent, use **FASIN**, **FACOS** and **FATAN** respectively. They return an angle in degrees or radians, depending on which is set. See Setting degrees or radians.

Calculating logarithms

Two words are supplied to calculate logarithms, **FLOG** and **FLN**. **FLOG** calculates a logarithm to base 10 (decimal). **FLN** calculates a logarithm to base e. Both take a floating point number in the range from 0 to ∞.

Calculating powers

Three power functions are supplied:

- e^x
- 10^x
- x^y

Calculating e^x

To calculate e^x, use **FE^X.** **FE^X** takes x as a floating point number.

Calculating 10^x

To calculate 10^x, use **F10^X**. **F10^X** takes x as a floating point number.

Calculating x^y

To calculate x^y, use **FX^Y**. **FX^Y** takes x and y as floating point numbers.

Setting degrees or radians

The angular measurement used in the trigonometric functions can be set to be either degrees or radians. To set it to degrees, use the word **DEGREES**. To set it to radians use the word **RADIANS**.

Converting between degrees and radians

To convert between degrees and radians use **RAD>DEG** or **DEG>RAD**. **RAD>DEG** converts an angle from radians to degrees. **DEG>RAD** converts an angle from degrees to radians.

Displaying floating point numbers

Two words are available for displaying floating point numbers, **F.** and **E.** . The word **F.** takes a floating point number off the stack and displays it in the form xxxx.xxxxx or x.xxxxxEyy depending on the size of the number. The word **E.** displays the number in the latter form.

Glossary

In the following glossary, you will find all the words that you are likely to need when using software floating point; the words omitted are, in general, subroutines used by words in the glossary.

N.B. Abbreviation: f.p. = floating point

D>F d — f
"d-to-f"

Converts a 32 bit double integer to a normalized f.p. number.

DEG>RAD f1 — f2
"deg-to-rad"

Convert f1 degrees to its corresponding number of radians.

DEGREES —
"degrees"

Switches floating point calculations to be done in degrees.

DINT f — d
"dint"

Leave the integer part of f as a double number on the stack.

DNORM d n — f
"d-norm"

Normalize double number d by n left shifts. Leaves a f.p. number on the stack.

E. f —
"e-dot"

Print the f.p. number on the stack in exponential form.

F, f —
"f-comma"

Compile the f.p. number on the top of the stack.

F. f —
"f-dot"

Print the top f.p. number on the stack in free format.

F! f addr —
"f-store"
Store the f.p. number f at address addr.

F+ f1 f2 — f3
"f-plus"
Add together the top two f.p. numbers on the stack and put the f.p. result on the stack.

F- f1 f2 — f3
"f-minus"
Subtract the top f.p. number on the stack from the second f.p. number on the stack, and put the f.p. result on the stack.

F* f1 f2 — f3
"f-star"
Take the top two f.p. numbers off the stack, multiply them together, and leave the f.p. result on the stack.

F/ f1 f2 — f3
"f-slash"
Divide the second f.p. number on the stack by the top f.p. number and leave the f.p. result on the stack.

F< f1 f2 — flag
"f-less-than"
Leave true flag if f1<f2. Otherwise, leave a false flag.

F<0 f — flag
"f-less-than-0"
Leave a true flag if f<0. Otherwise, leave a false flag.

F= f1 f2 — flag
"f-equals"
Leave a true flag if the top two f.p. numbers on the stack are equal. Otherwise leave a false flag.

F0= f — flag
"f-0-equals"
Leave a true flag if the f.p. number on the top of the stack is zero.

F> f1 f2 — flag
"f-greater-than"
Leave a true flag if f1>f2. Otherwise, leave a false flag.

F>0 f — flag
"f-greater-than-zero"
Leave a true flag if the f.p. number on the top of the stack is greater than zero.

F# — f [executing]
"f-hash" — [compiling]
If interpreting, takes text from the input stream and, if possible, converts it to a f.p. number on the stack. Numbers in integer format will be converted to floating point. If compiling, the converted number is compiled.

F#IN — f 3 | 0
"f-hash-in"
Attempts to convert a token from the input stream to a floating point number. Numbers in integer format will be converted to floating point. An indicator (0 or 3) is returned in the same way as an indicator is returned by **FNUMBER?**.

F@ addr — f
"f-fetch"
Fetch the f.p. number from address addr and put it on the stack.

F10^X f1 — f2
"f-10-to-the-x"
Raise 10 to the power f1 and put the result on the stack.

FABS f — |f|
"f-abs"
Returns the modulus of the f.p. number on the top of the stack.

FACOS f1 — f2
"f-a-cos"
Leave, on the stack, the angle (in degrees or radians) whose cosine is f1, such that 0<=f2<=180 (f2 in degrees).

FARRAY fn-1..f0 n — [parent]
"f-array" n — fn [child]

When generating the array, take n f.p. numbers and n, and compile them into the array. When executing the child word, take n and place f.p. number n from the array onto the stack. Note that the numbering in the array goes 0,1,..n-1.

FASIN f1 — f2
"f-a-sine"

Leave, on the stack, the angle (in degrees or radians) whose sine is f1, such that -90<=f2<=90.

FATAN f1 — f2
"f-a-tan"

Leave, on the stack, the angle (in degrees or radians) whose tangent is f1, such that -90<f2<90.

FCONSTANT f — [parent]
"f-constant" — f [child]

Floating point equivalent of **CONSTANT**. Use in the form:

<f.p. number on stack> FCONSTANT <name>

FCOS f1 — f2
"f-cos"

Take the cosine of f1 (degrees or radians) and put it on the stack.

FDROP f —
"f-drop"

Drop the f.p. number on the top of the stack.

FDUP f — f f
"f-dup"

Duplicate the f.p. number on the top of the stack.

FE^X f1 — f2
"f-e-to-the-x"

Raise e, the exponential number, to the power f1 and put the result on the stack.

FFRAC f1 f2 — f3
"f-frac"

Leave the fractional remainder from the division f1/f2. The remainder takes the sign of the dividend.

FINT f1 — f2
"fint"

Place the f.p. integer value of f1 on the stack.

FLITERAL f —
"f-literal"

When compiling, compile f as a literal. For example,

```
: ABCD  [ calculate f ] FLITERAL ;
```

Compilation is suspended for the compile-time calculation of f. Execution of **ABCD** leaves f on the stack.

FLN f1 — f2
"f-log-base-e"

Take the logarithm of f1 to base e and put the result on the stack.

FLOATS —
"floats"

Switches the action of **NUMBER?** to be **FNUMBER?**. This action can be reversed by **INTEGERS**. Both **FLOATS** and **INTEGERS** are in the **FORTH** vocabulary.

FLOG f1 — f2
"f-log-base-10"

Take the logarithm of f1 to base 10 (decimal) and put the result on the stack.

FMAX f1 f2 — max{f1,f2}
"f-max"

Put the greater of the top two f.p. numbers onto the stack.

FMIN f1 f2 — min{f1,f2}
"f-min"

Put the lesser of the top two f.p. numbers onto the stack.

FNEGATE f — -f
"f-negate"

Negate the f.p. number on the top of the stack.

FNUMBER? addr — 0 | n 1 | d 2 | f 3
"f-number-query"

Converts string at address addr to either a single, double or floating point number along with 1, 2, or 3 respectively. If a 0 is left on the stack then **FNUMBER?** was unable to convert the string.

FOVER f1 f2 — f1 f2 f1
"f-over"

Floating point equivalent of **OVER**.

FROT f1 f2 f3 — f2 f3 f1
"f-rote"

Floating point equivalent of **ROT**.

FSEPARATE f1 f2 — f3 f4
"f-separate"

Leave the signed integer quotient f4 and remainder f3 when f1 is divided by f2. The remainder has the same sign as the dividend.

FSIGN f — f flag
"f-sign"

Leave the f.p. number and a flag on the stack. Leaves a true flag if f is negative, else leaves a false flag.

FSIN f1 — f2
"f-sine"

Leave the floating point sine of f1 (degrees or radians) and put it on the stack.

FSQR f1 — f2
"f-s-q-r"

Take the square root of the floating point number on the top of the stack and put the result onto the stack.

FSWAP f1 f2 — f2 f1
"f-swap"

Floating point equivalent of **SWAP**.

FTAN f1 — f2
"f-tan"

Take the tangent of f1 (degrees or radians) and put the result on the stack.

FVARIABLE —
"f-variable"

Floating point equivalent of **VARIABLE**. Set up an fvariable by typing:

FVARIABLE <name>

FX^N f1 n — f2
"f-x-to-the-n"

Raise f1 to the power n (n integer), and put result on the stack.

FX^Y f1 f2 — f3
"f-x-to-the-y"

Raise f1 to the power f2 and put the result on the stack.

INTEGERS —
"integers"

Switches the action of **NUMBER?** to be **INTEGER?**. This action reverses that of **FLOATS**. Both **FLOATS** and **INTEGERS** are in the **FORTH** vocabulary.

RAD>DEG f1 — f2
"rad-to-deg"

Convert f1 radians to degrees, and put result on the stack.

RADIANS —
"radians"

Switches floating point calculations to be done in radians.

S>F n — f
"s-to-f"

Converts a single (16 bit) number to a normalized f.p. number

SINT f — n
"sint"

Takes the single number integer part of f and puts it on the stack.

Blank page

ROM PowerForth Utilities

Supplied as source are utilities to:

- compile source code files on your target board
- upload a binary image from your target to your PC

These utilities can be used to generate an EPROM which has all the tools required to develop an application.

Compiling text files

Source text files can be compiled from the host PC onto the target system. This saves time in not having to cross compile all the source if a small modification is made. The utilities assume that each text file is split into pages. A page is separated from another by an ASCII 12 character. Writing source code with pages gives you the ability to compile discrete chunks of code. If you do not have any pages in your source code, the whole file should be treated as page one.

Note: You must switch XShell to file server mode to use this facility. See XShell manual.

The required files

To compile text files from your target board, cross compile the files IODEF.FTH and TEXTFILE.FTH.

Compiling a specified text file

To compile all or part of a specified text file onto your target, use FROM-FILE in the form:

start-page end-page FROM-FILE <name>

This compiles the file **<NAME>** into the target's dictionary.

Compiling the default text file

An alternative approach is to specify a default filename which is remembered by the target. The file can then be compiled without specifying the text file's name. This is normally quicker if you are continuously compiling one file.

Specifying the default text file

To set the default filename, type:

USE <name>

where **<NAME>** is the text file's name to be set as the default. If no extension is specified, an extension of **.**FTH is assumed.

Compiling the default text file

To compile the default file, type:

start-page end-page FROM

This compiles the pages from start-page to end-page onto the target.

Specifying the start and end pages

Words are supplied to enable you to compile parts or all of a file easily. To compile parts of a file you can use **ONWARDS**, **UPTO** and **ALONE** with either **FROM** or **FROM-FILE**.

Compiling from a specified page to the end of the file

To compile from a start page to the end of the file, use **ONWARDS** in the form:

start-page ONWARDS

This generates a start and end page which can be used with either **FROM-FILE** or **FROM**. For example,

10 ONWARDS FROM

compiles from page ten to the end of the default text file.

Compiling from the start of the file to a specified page

To compile from the start of a file to a specified page, use **UPTO** in the form:

end-page UPTO

This generates a start and end page which can be used with either **FROM-FILE** or **FROM**.

For example,

10 UPTO FROM

compiles from the start of the file to page ten of the default text file.

Compiling a single page

To compile a single page, use **ALONE** in the form:

page# ALONE

This generates a start and end page which can be used with either **FROM-FILE** or **FROM**.

For example,

10 ALONE FROM

compiles page ten of the default text file.

Compiling the whole file

To simplify the compilation of the whole file, use the word **ALL**:

ALL FROM

ALL FROM-FILE <name>

Compiling screen files

Standard Forth screen files can be compiled onto the target system, in the same way as on a host system.

Note: You must switch XShell to file server mode to use this facility. See XShell manual.

The required files

To compile screen files from your target board, cross compile the files IO-DEF.FTH and BLOCKS.FTH.

Compiling a specified screen file

To compile all or part of a specified screen file onto your target, use **THRU-USING** in the form:

start-screen end-screen THRU-USING <name>

This compiles the file **<NAME>** into the target's dictionary.

Compiling the default screen file

An alternative approach is to specify a default filename which is remembered by the target. The file can then be compiled without specifying the screen file's name. This is normally quicker if you are continuously compiling one file.

Specifying the default screen file

To set the default filename, type:

USING <name>

where **<NAME>** is the screen file's name to be set as the default. If no extension is specified, an extension of **.**SCR is assumed.

Compiling the default screen file

To compile the default file, type:

start-screen end-screen THRU

This compiles the screens from start-screen to end-screen onto the target.

Compiling a single screen

A single screen can be loaded from the default screen file or a specified screen file.

Compiling a single screen from a specified screen file

To compile a single screen, use **LOAD-USING** in the form**:**

screen# LOAD-USING <name>

This compiles the screen screen# of the file **<NAME>** onto the target.

Compiling a single screen from the default screen file

To compile a single screen, use **LOAD** in the form**:**

screen# LOAD

where screen# is the screen number to load.

Downloading a binary image

A binary image can be downloaded from the target to your host PC. Two utilities are provided:

- an Intel hex download
- an XMODEM download

For both utilities a suitable communications package will be required (e.g. ProComm).

XMODEM binary image download

Binary images can be downloaded to your PC using the XMODEM protocol.

Required files

To use this utility you must cross compile the files BLOCKS.FTH and BIN-DOWN.FTH.

Using the XMODEM binary download utility

To down-load a binary image from the target system to your PC, use BIN-DOWN in the form:

```
addr #bytes BIN-DOWN
```

where addr is the start address and #bytes is the number of bytes to down-load starting from addr.

For example,

```
1200 400 BIN-DOWN
```

sends the area of memory from 1200 to 1599 to your host PC.

Intel hex download

Memory images can be downloaded to your PC using the Intel hex format.

Required files

To use this utility you must cross compile the files BLOCKS.FTH and HEX-DOWN.FTH.

Using the hex download utility

To download a hex image from the target system to your PC, use HEX-DOWN in the form:

```
addr #bytes HEX-DOWN
```

where addr is the start address and #bytes is the number of bytes to down-load starting from addr.

For example,

1200 400 HEX-DOWN

sends the area of memory from 1200 to 1599 to your host PC.

ROM PowerForth

ROMPowerForth can be used to generate a stand-alone Forth system. With these utilities, you can generate an EPROM which contains an interactive Forth with the ability to develop an application.

Note: A licence is required to distribute open Forth systems. Contact MPE for more details.

Hardware requirements

To develop an application using ROM PowerForth, your board requires three areas of which:

- one is always EPROM
- one is always RAM
- one is RAM for development and EPROM for application

EPROM area

The area which is always EPROM, contains the development kernel.

RAM area

The area which is always RAM is used for variables and all changeable data.

RAM/EPROM area

This area is used to develop your application. Therefore, it must be RAM while developing. Once your application is developed, the application's image must be saved into battery-backed RAM or EPROM. Therefore, this area must have the ability to be alterable but also non-volatile.

Types of board

The type of board that can be used to develop using ROM PowerForth is restricted to:

- three site boards
- two site boards with battery backed RAM
- two site boards with socket converter

Three site boards

The three areas are provided by three memory sockets:

- EPROM holding development kernel
- RAM which holds the variables and changeable data
- EPROM or RAM which is selectable by a link on the board

Two site boards with battery backed RAM

The three areas are provided by two sockets:

- EPROM holding the development kernel
- battery-backed RAM which is split into two areas

Two site boards with socket converter

On many boards, there is unused space in the EPROM as ROM PowerForth occupies less than 16k bytes of memory. Therefore, a header board can be made which converts one socket into two. For example, if the socket normally takes a 27256 EPROM, a board can be made which has a 16k EPROM with the ROM PowerForth development kernel and 16k bytes of RAM. To access the RAM, the write line is attached to a suitable point on the main board with a flying lead.

After the application has been developed, the two images are combined back into a single EPROM.

Making your application turnkey

Once your application has been developed, it needs to be made turnkey so that it is always available. The application can be made semi-permanent by compiling into battery-backed RAM in the RAM/EPROM area. Alternatively, it can be copied into an EPROM if the board allows.

Configuring a turnkey application

The word SETUP takes the address of the word passed to it and marks this in the RAM/EPROM header as the address of the word to be run at power-up. If a value of zero is passed to SETUP, the interactive Forth kernel will be run at power-up.

For example, the word JOB is to be run at power-up. Therefore you type,

```
‘ JOB SETUP
```

Discarding the application RAM area

The application can be discarded by typing:

```
0 ROM !
```

Changing the application RAM start address

The constant ROM returns the start address of the application RAM area. If the address of this area is to be changed, the EPROM must be modified. To do this, the 16-bit value in ROM must be changed.

Glossary

$FROM-FILE first last $addr —
"dollar-from-file"

Compiles a text file given by the counted string $addr. Pages from first to last will be compiled. e.g.

10 20 "" TEST.FTH" $FROM-FILE
Compiles pages ten to twenty of file TEST.FTH.

$USING addr$ —
"dollar-using"

Sets the default screen file to the counted string addr$.

"" TEST.SCR" $USING
sets the default screen file to TEST.SCR.

ALL — first last
"all"

Used with FROM and FROM-FILE to compile a complete file.

ALL FROM
compiles all of the default text file.

ALONE n — first last
"alone"

Used with **FROM** and **FROM-FILE** to compile a single page.

1 ALONE FROM
compiles page one of the default text file

CLS —
"c-l-s"

Clears the display.

EMPTY-BUFFERS —
"empty-buffers"

Marks screen file buffers as empty

FLUSH —
"flush"

Flushes the screen file buffer to disk

FROM first last —
"from"

Compiles pages first to last of the default text file.

FROM-FILE first last <name>—
" from-file"

Compiles a range of pages (first to last inclusive) from a specified text file <name>.

INDEX n1 n2 —
"index"

List top lines in range of screens.

L —
"l"

Displays the current screen.

LIST blk# —
"list"

Display screen given.

LOAD blk# —
"load"

Compile given screen.

LOAD-USING blk# <screen-file> —
"load-using"

Compiles the given screen in the specified screen file.

N —
"n"

Displays the next screen.

ONWARDS first — first last
"onwards"

Used with **FROM** and **FROM-FILE** to compile from a specified page to the end of the file.

P —
"p"

Displays the previous screen.

QX —
"q-x"

Displays the top line of every screen in the default screen file.

SAVE-BUFFERS —
"save-buffers"

Saves the screen file buffers if they have been modified.

SET-USEFILE $addr —
"set-use-file"

The counted string $addr is set to be the default screen file

THRU blk#from blk#to —
"thru"

Compiles from screens blk#from to blk#to (inclusive) of the default screen file

THRU-USING blk#from blk#to <screen-file> —
"thru-using"

Compiles from screens blk#from to blk#to (inclusive) of the specified screen file.

UPDATE —
"update"

Flags the screen file buffer as being modified.

UPTO last — first last
"upto"

Used with **FROM** and **FROM-FILE** to compile from the start of the file to the specified page.

10 UPTO FROM
compiles from the start of the default text file to page 10.

USE — ;<name>
"use"

Specifies the default text file. If an extension is not included in the filename, .FTH is assumed.

USE TEST.FTH
sets the default screen file to TEST.FTH.

USING — ; <name>
"using"

Specifies the default screen file. If an extension is not included in the filename, .SCR is assumed.

USING TEST.SCR
sets the default text file to TEST.SCR.

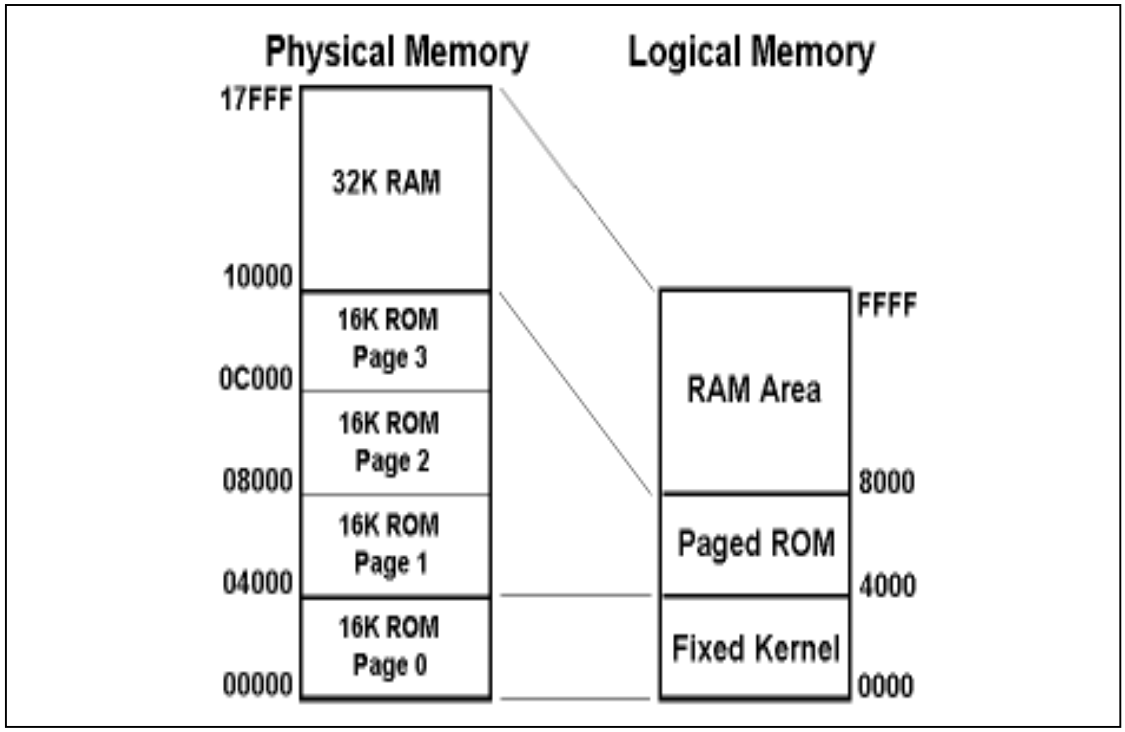
Blank Page

Paged targets

Many people develop for 8-bit and 16-bit target processors. One disadvantage of using some 8-bit and 16-bit processors is their limited addressing range (64KB). To overcome this limitation the MPE cross compiler supports paged targets. Paging is used when the application's size is larger than the available memory. To overcome this, different parts of the application are loaded into memory when required.

An example memory map for the 64180 is shown in figure 11. The memory space is split into three sections:

- Kernel - The Forth kernel. It is always present. It maps into the bottom 16k of the 27512 EPROM.
- ROM - The target's ROM is mapped into 16k chunks of the 27512. This gives three further pages of ROM.
- RAM - The fixed data RAM. This holds all the initialised variables.



□□□□□□□ 11 □ □□□□□□□□ □□□□□□□ □□□□□□□□□□

Creating a paged target

To create a paged target, you need to define each page's memory map and write the page switching word. You may also need to initialise the CBAR CBR and BBR registers if you are using a 64180.

Initialising a 64180 for paging

In order to use the on board MMU of the 64180 for paging it is necessary to set up the CBAR, CBR and BBR registers as early as possible in the initialisation sequence. Typically this should be done during startup code, before the Forth starts. To initialise the above memory map, you should write the value 084h into the CBAR register, 08h into the CBR register and 00h into the BBR register in order for the MMU to correctly decode the addresses. Although the BBR register defaults to 00 on power up, it is good practice, although not strictly necessary, to initialise it explicitly.

Defining a page

A page is defined in a similar way to the kernel. Seperate pages for code and data can be defined. A page is defined by three items:

- the start of the page in addressable space
- the end of the page in addressable space
- a unique identifier for the page.

To define a code page, use **CODE-PAGE**. To define a data page, use **DATA-PAGE**. **CODE-PAGE** is used in the form:

<start> <end> <page-id> CODE-PAGE <name>

DATA-PAGE is used in the form:

<start> <end> <page-id> DATA-PAGE <name>

where <start> is the start address in the page, <end> is the last address in the page and <page-id> is the page's identifier. The <page-id> must uniquely identify the page, as it is used to indicate which page to switch to.

For example, to define the three pages in figure 11, you code:

```
$4000 $7FFF 0 CODE-PAGE Page1
$4000 $7FFF 4 CODE-PAGE Page2
$4000 $7FFF 8 CODE-PAGE Page3
```

```
: PAGE@       \ — page-id ; returns current page
  BBR P@                    \ value of BBR register
;

: PAGE!       \ page-id — ; switch to a new page
  BBR P!                    \ write BBR register
;

\ switch to page and execute word
: PAGE-WORD                 \ cfa page-id — ;
  PAGE@ >R                  \ preserve current page
  PAGE!                                 \ switch to new page
  EXECUTE                               \ execute word within page
  R> PAGE!                  \ restore previous page
;
```

□□□□□□ 13 □ □□□□□□□ □□□□ □□□□□□ □□□□

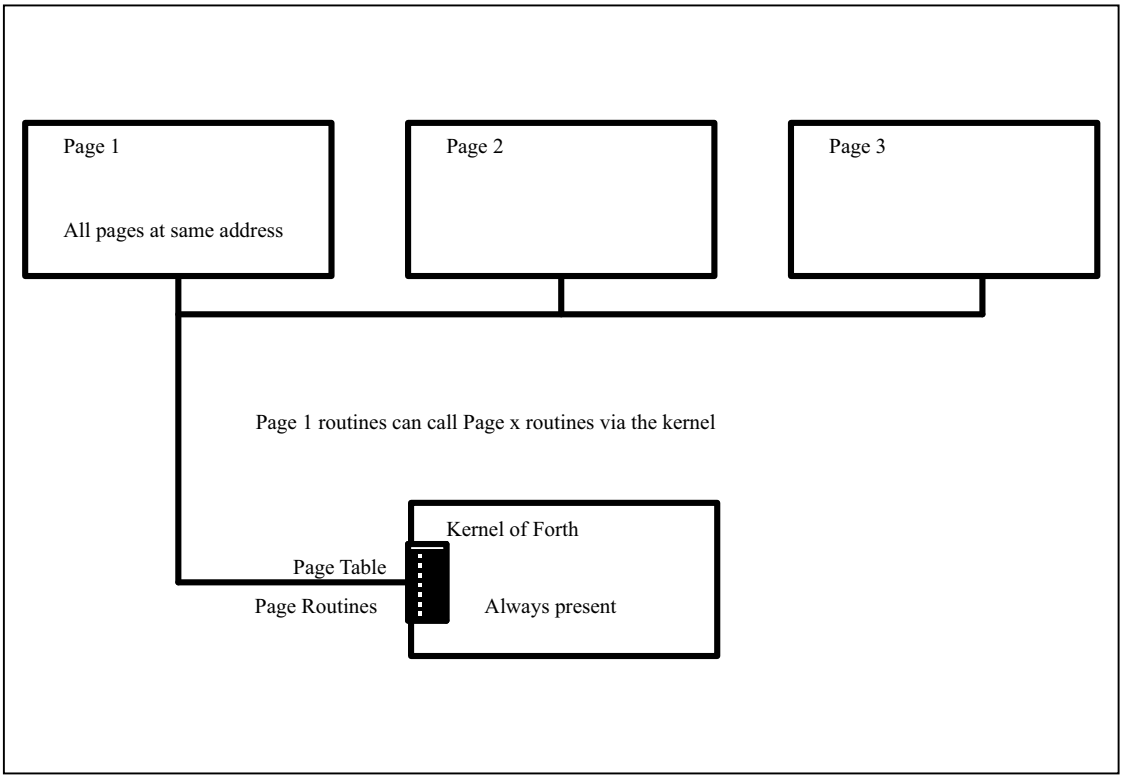
The page-id used is a unique identfier which is meaningful to the page switching word. On the 64180 this value will be placed directly in the BBR register in order to perform the page switch. Note that since the page-id must be unique you should not use 0 as the page-id for the KERNEL-RAM area.

Writing the page switching word

A word is required to do the actual hardware page switch. The page switching word **must** be called **PAGE-WORD** and must be compiled in the fixed kernel part of the target image. When a word in a different page is to be executed from a page, the actual execution must be performed via **PAGE-WORD**. This is shown in figure 14. The cross compiler will automatically compile a reference to **PAGE-WORD** every time an external page is referenced. **PAGE-WORD** is passed the page-id of the required page and the address of the word to be executed within the page. **PAGE-WORD** must:

- preserve the current page id
- switch the specified page into addressable memory
- execute the required code within the page
- restore the previous page

An example is shown in figure 12. Once tested, this can then be coded in assembler for speed.



□□□□□□ 14 □ □□□ □□□□ □□□□□□□□□ □□□□□□□□□

Compiling code into a page

Compiling your source code is very similar to compiling code into the kernel, but some extra initialisation must be done and some restrictions must be observed.

Initialising compilation into a page

The majority of Forth can be compiled into a page except for inter-page deferred words.

To interactively debug your paged code, the cross compiler requires it to be placed in a paged vocabulary. To do this, define a paged vocabulary,

<start> <end> <id> CODE-PAGE <name>
PAGED-VOCABULARY <name>

where <name> is the name of a page.

Compiling into a page

To compile into a page use:

USE-CODE <name>

where <name> is the name of the page you wish to compile into. Any code compiled after this instruction will be compiled into the page <name>. You can switch between compilation pages at any time, so that all your code for one page does not need to be compiled together.

Restrictions for compiling into a code page

You cannot forward reference a word in a different page.

Finishing compilation of a page

Once your code has been compiled into a page, **FINIS-CODE-PAGE** must be executed, in the form:

FINIS-CODE-PAGE

The cross compiler shows a compilation summary for the page.

Note: The label **INIT-RAM** will be shown as an unresolved reference. This is correct behaviour as the cross compiler uses this label internally.

Compiling data into a page

Compiling data such as variables and constants into a page is straightforward but some restrictions must be observed.

Setting the data page

To select the page that data is to be used for, use **USE-DATA** in the form:

USE-DATA <name>

where <name> is the name of the page defined with **DATA-PAGE**. Variables and constants can then be defined.

Restrictions for compiling into a data page

Data defined in pages other than the kernel page will not have their data initialised. This must be done at startup by the application.

The **KERNEL-RAM**‘s page-id must be set to the **KERNEL**‘s page-id.

Controlling the compiler

While cross-compiling, the cross-compiler needs to be instructed on how to configure itself. You need to tell the cross compiler:

- when to start compiling
- when to stop compiling
- whether to enable floating point
- whether to turn the compiler log on or off
- which code and data page to compile into
- selectively compile portions of code

These instructions are normally placed in the control file, before any instructions are compiled.

Starting the cross-compiler

To start cross compiling, use the word **CROSS-COMPILE**. Any code after this directive will be cross compiled into the target image instead of compiled onto the cross compiler.

Stopping the cross-compiler

To stop the cross compiler cross-compiling, use **FINIS**. **FINIS** stops cross compiling, closes all files and returns to XShell.

Enabling floating point

If you want the compiler to be able to handle floating point numbers, you need to instruct it with the word **FLOATS**. The default is integer only.

Turning the log on and off

The cross-compiler log can either display dots (when off) or information on the items compiled (when on). To turn the log on, use **LOG**. To turn the compiler off, use **NO-LOG**.

Selecting code and data page

In a paged system you need to select which pages code and data are compiled into. To do this use **USE-CODE** and **USE-DATA**. They are used in the form:

USE-CODE <name>
USE-DATA <name>

where <name> is the name of the page to compile code into. <name> was specified when defining the memory map using **KERNEL** and **KERNEL-RAM**.

Conditional compilation

Conditional compilation is used to selectively compile portions of code. Three words are available to do this, **IF(**, **)ELSE(** and **)ENDIF**. These are analogous to **IF**, **ELSE** and **ENDIF**. They can be used within Forth words to selectively compile portions of it, or can be used outside a Forth word to selectively compile whole words.

An example

Two code examples are shown in figures 16 and 17. The examples given perform conditional compilation inside and outside a colon definition.

```
ONLY FORTH ALSO C-C DEFINITIONS  \ Switch vocabularies

CC/C                    \ Switch to compiler vocabulary
3 CONSTANT 3OR4?        \ add the word 3OR4?

ONLY FORTH ALSO C-C DEFINITIONS  \ Restore search order
```

□□□□□□ 15 □ □□□□□□ □□□□□ □□ □□□ □□□□□□□□

Conditional compilation outside a colon definition

The example shown in figure 16 compiles one of the **PRINT1OR2**'s. Which one is compiled is dependant on the value of **1OR2?**. If it is set to one, **PRINT1OR2** displays a one when executed. If it is set to two, **PRINT1OR2** displays a two.

Conditional compilation within a colon definition

Using conditional compilation within a colon definition is slightly more complicated. This is because you need to write a word which places a number on the cross-compiler's stack when it's cross-compiling. An example is shown in figure 15, where a constant **3OR4?** is added to the compiler. This can then be used in the example in figure 17.

```
1 EQU 1OR2?

1OR2? 1 =       \ Display one or two?
IF(             \ If 1OR2?=1, PRINT1 will be compiled
: PRINT1OR2 \ — ; Display a one
  ." 1"
;
)ENDIF
1OR2? 2 =
IF(             \ If 1OR2?=2, PRINT2 will be compiled
: PRINT1OR2 \ — ; Display a two
  ."2"
;
)ENDIF          \ End marker for conditional compilation
```

□□□□□□ 16 □ □□□□□□□□□□□□ □□□□□□□□□□□□ □□□□□□□□

```
: PRINT3OR4 \ — ; Display a three or four
  3OR4? [ 3 = ]                                  \ compiler word
  IF(  ." 3"  )ENDIF              \ Display a three
  3OR4? [ 4 = ]
  IF(  ." 4"  )ENDIF              \ Display a four
;
```

□□□□□□ 17 □ □□□□□□□□□□□ □□□□□□□□□□□ □□□□□□□

Forth on the Target

This chapter describes how a Forth is laid out on a target board. It is therefore not necessary to read this chapter, but this chapter provides more information if you are interested or if you want to perform more advanced modifications to the cross-compiler or target.

Inside a ROM target Forth

A standalone ROM target Forth communicates with the host via a serial line. The host needs to be running a dumb terminal emulator. The terminal emulator displays any characters which arrive from the target and sends any characters typed at the host's keyboard. The target takes input and makes output directly from the serial line, not from a keyboard and to a display. To do this, the deferred words **EMIT** and **KEY** have the actions **SER-KEY** and **SER-EMIT** respectively.

The Forth memory map

A typical Forth system consists of areas as in figure 18. The RAM on the target system is split into several areas:

- a user area for interrupts
- a user area and stacks for eack task
- a terminal input buffer (TIB) for the forth

The remaining RAM is available for use by the Forth as dictionary space.

RAM Initialisation

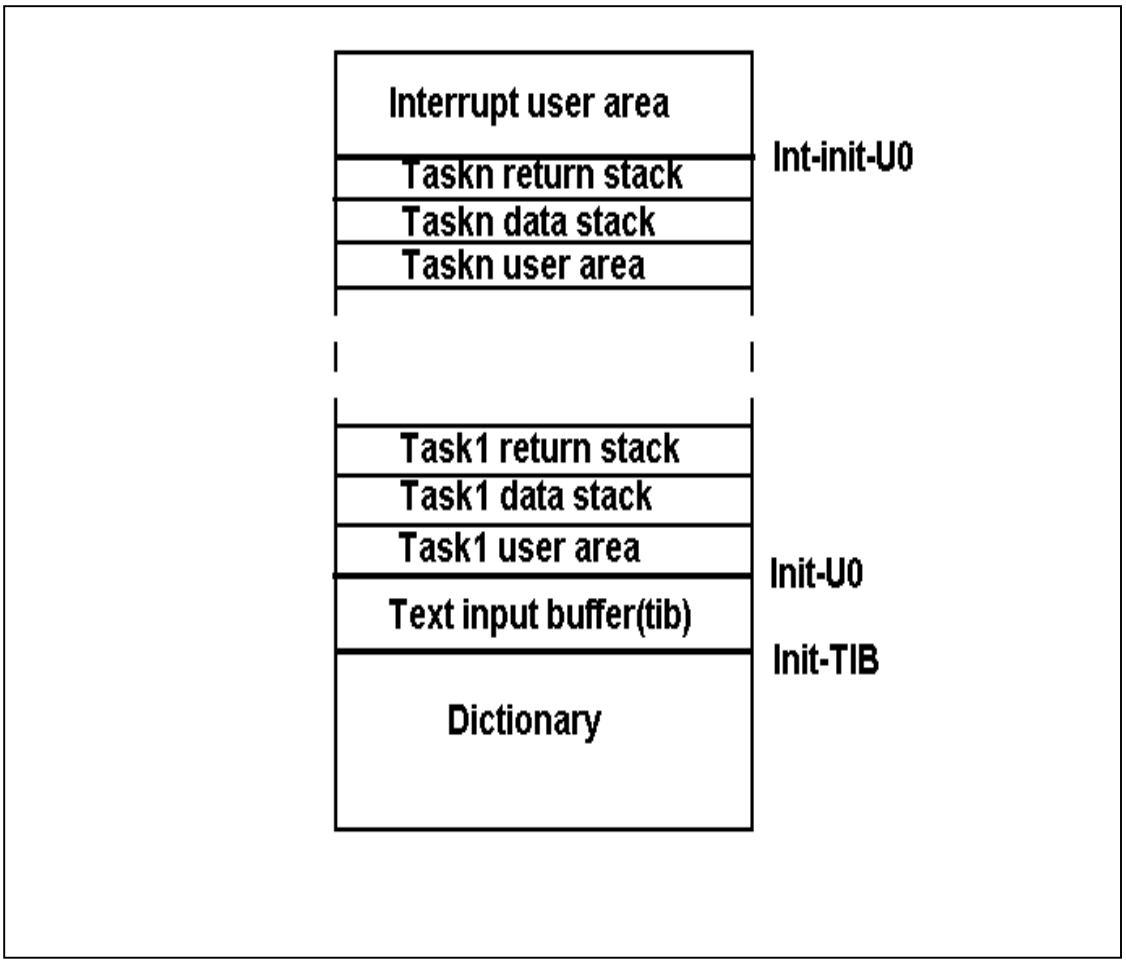
The Forth-83 standard does not require variables (created by words **VARIABLE** or **CVARIABLE**) to be automatically initialised at start up. In MPE

PowerForth variables are initialised to zero within the cross-compiler. The table of initial values is then copied to the end of the output file when the cross-compiler terminates.

Two locations in the target, **ROM-TABLE** and **RAM-TABLE**, point to the initial value table (in ROM), and to the memory area (in RAM) it should be copied to. The first two bytes of the table contain a count of the number of bytes to be copied. The code that performs this copy can be found in the word **(INIT)** in KERNEL.FTH.

The maximum size of the initial value table is set within the cross-compiler to 512 bytes. To alter the size, alter the parameter for the **KERNEL-RAM** directive in the control file, and recompile.

RAM can also be initialised when space is allotted by the cross-compiler words **C,(R)** or **,(R)**. It is safest to explicitly initialise all variables and data areas in **COLD** or **ABORT**. This protects the system from errant behaviour after error recovery or power failure. It is worth remembering that a Mariner probe was lost because of an uninitialised Fortran variable!



□□□□□□ 18 □ □□□ □□□□□ □□□ □□□□□□ □□□

Register Usage

The 64180 Forth implementation uses direct threaded code for speed. The assignment of the 64180 registers is as follows:

64180 register Forth	Function register	
BC	IP	Forth interpretive pointer
SP	SP	Data stack pointer
IY	RP	Return stack pointer
UP (in DP RAM)	UP	User Pointer

In addition, the IX register may be used by the NEXT, macro, depending on which version of the macro you have chosen to use. The HL register points to the CFA on entry to a CODE definition, so if you wish to make use of this fact, you will need to preserve this value before using the HL register.

Direct threading

For speed, PowerForth uses Direct Threaded Code as it is a good compromise between speed and space. The routine which threads between the Forth words is called **NEXT,** and will be found defined in the files CODEZ80.FTH and CODE180.FTH.

All Forth words except for **CODE** words contain a JP (absolute jump) instruction in the code field. On entry to the code portion of a defining word the CFA (Code Field Address) will available in the HL register. See the example of **VARIABLE**.

Forth Models

Two different targets are provided in the ROM and CHIP directories. The ROM directory contains a standalone Forth which can be debugged interactively using a dumb terminal or XShell3. All the facilities you need are provided by the Forth. Source code can be downloaded to the Forth and debugged on the target. Interpretation and compilation are provided by the target Forth.

The CHIP directory contains a Forth that is tuned for single chip applications. Unlike the Forth in the ROM directory, the Forth in the CHIP directory requires the Umbilical Forth message passer in the TARGEND.FTH file for interpretation and compilation, which is provided by a server on the host PC (see below). The CHIP directory contains a system that contains a fully interactive Forth kernel in less than 3k bytes.

All directories use the same implementation model, and so code from one system can be used by another. Thus an application using the CHIP directory as a basis, can safely use code from the ROM directory.

Inside Umbilical Forth

Umbilical Forth behaves in the same way as a ROM target Forth, but the mechanism that provides the interaction with the target is totally different.

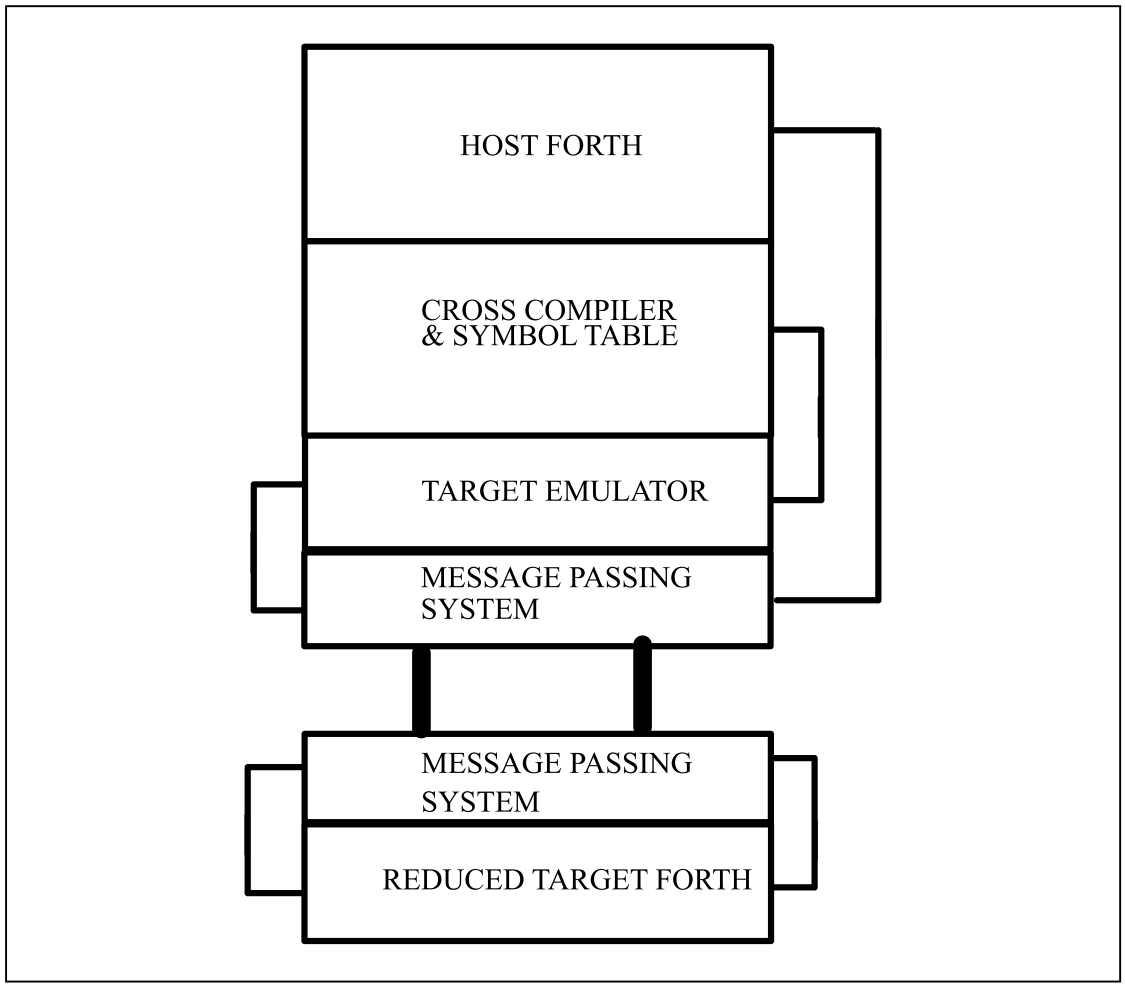


Figure 19 - Umbilical forth message passing

When you reset the target and the board signs-on, you are still running the cross compiler. Umbilical Forth is therefore an extension of the cross compiler to provide interactive cross interpretation and cross compilation.

When a word is cross compiled, the cross compiler places information in the symbol table. The symbol table therefore contains the CFA of the word in the target image. By using a message passing system between the cross-compiler and the target, the CFA of the word can be passed to the target. The host can then execute the word on the target, passing parameters to and from as appropriate. Therefore, the target does not need any headers in the target image, nor does the target need any of the code to process the headers.

Blank Page

Optimizing your development cycle

While developing an application, you cycle through a series of steps:

- editing your source code
- cross-compiling to generate a binary image file
- downloading to an EPROM emulator/programmer
- testing and debugging your code

This development cycle is repeated until all development and debugging is completed. The quicker you can go round this cycle, the quicker your application is finished. XShell and the cross compiler help you achieve these aims.

Speeding up the compilation

Every time a cross compilation is carried out, certain sections of code, which are never altered, are compiled again and again. This is particularily the case for the kernel files which generate the Forth image. You can use the partial compilation feature of the cross compiler to halt the cross compilation at a strategic position and save the cross compiler's state. You can then continue cross-compiling from this saved position. In this way, you can dramatically reduce the time the application takes to compile.

Note: Partial compilation cannot be used when directly compiling to an emulator

Saving the compilation state

To stop and save the cross-compilation at a required place, use **SUSPEND**. **SUSPEND** is used in the form:

SUSPEND <filename>

where <filename> is the name of files the cross compiler will use to save the state information. The filename is a name **without** an extension.

Restarting from a saved state

To restart from a previously saved cross-compilation state, use **RESTART**. **RESTART** is used in the same form as **SUSPEND**,

RESTART <filename>

where <filename> is the filename used when saving the compilation state. **RESTART** must be used instead of the word **CROSS-COMPILE** and any macros must be loaded again.

Note: The image file created by the compiler after a **SUSPEND** must exist in the compilation directory.

An example

An example control file can be found in the directory ROM\PARTIAL.

Speeding up the download

The cross compiler has the facility to download the image to the LeBurg emulator while it is compiling. This speeds up the turn-around of the edit, compile, download and test cycle by removing the download step. To download directly to a LeBurg emulator, you need to tell the cross compiler:

- what size of EPROM it is generating
- the bus width (e.g. 8 bit, 16 bit)
- which page to put in the emulator

You also need to load the driver TSR for your emulator before running the cross compiler.

Target bus width	width
8	8bit
16	16bit
32	32bit

Table 8 - Available bus widths

Table 9 - Available EPROM sizes

Note: This facility cannot be used with partial compilation.

Setting EPROM size and bus width

To set the size of EPROM to use and the bus width of the target board, use **OUTPUT-EMULTOR**. This is in the form:

size width OUTPUT-EMULATOR

where size and width can be selected from tables 8 and 9. If you are using a larger emulator, you may choose from:

e2764 e27128 e27256 e27512 e27010 e27020 e27040 e27080

For example, if your board uses a 27256 and your target has an 8-bit bus width, code:

e27256 8bit OUTPUT-EMULATOR

This instruction must be placed in your control file **before** the **CROSS-COMPILE** directive.

Setting the page

To send a page to an EPROM emulator, use **IN-EMULATOR** in the form:

xxxx IN-EMULATOR <name>

where <name> is the name of the page set by **KERNEL** or **CODE-PAGE**and xxxx is the base address in the emulator where to place the image.

Using the emulator driver

To download to the emulator you first need to load an emulator driver. These are in the directory X180\EMU-TSR. Which TSR you use depends on the emulator you have. If you have the LePROM, use the file TSR021.COM. If you have the LeBIG or the LeMEG then use the TSR041.COM.

One of these emulator drivers should be loaded before you enter XShell. It is normally convenient to load them from your AUTOEXEC.BAT file.

Technical glossary

Compiler log When each label, variable, constant or colon defintition is cross compiled the cross compiler displays a dot or information about the compiled item.

Control file A file which is loaded by the cross-compiler. It contains directives to the cross-compiler and the names of any additional files to be compiled.

Cross compiler A program which generates executable code for a processor different to which it is running on.

Dictionary A list of words defined in a Forth system

Event A non-regular occurence. In the multitasker an event is used to trigger a task.

Glossary A list of forth words with their pronunciation, stack effect and a brief description of their action.

Host The platform the cross compiler runs on. Normally a PC.

Host mode One of XShell's modes which is a Forth for the PC.

Image file The output of the cross compiler. It has the extension .IMG by default.

Initialised RAM See RAM table.

Kernel The code required for interactive Forth.

Memory map A description of the start and end of ROM and RAM in memory

Multitasker A program which allows a processor to run more than one task by continuously switching between different tasks.

Paged target A system where there is more memory available that can be addressed at one time. Areas of memory can be switched into an addressable range, so simulating a larger address space than is physically possible.

RAM table An area of memory in the ROM that is copied to RAM at startup. It contains any initial values of variables.

ROM target forth A Forth which works on a ROM/RAM system as opposed to a RAM system.

ROM/RAM target A target with code executed out of ROM and data kept in RAM.

Scheduler The part of a multitasker which switches to the next task

Screen file A type of file which Forth source was originally developed in.

Serial line driver The words which interface the target code to the serial line. These are device dependant whereas the rest of the kernel is generic.

Symbol table Used and generated by the cross-compiler. It contains information on each item compiled.

Target The processor or board that the cross compiler is generating code for.

Target mode One of XShell's modes which acts as a dumb terminal. It lets you communicate with your target board.

Task In a multitasking environment, a task is a stand-alone program which appears to run simultaneously with other tasks.

Task control block Where information about a task is kept. It is used by the scheduler to switch to the next task.

TCB See task control block

UART Universal Asynchronous Receiver/Transmitter - Sends and receives serial data.

Umbilical Forth A reduced Forth designed for single chip targets. Uses a message passing system to commicate with the host.

Unresolved references Any words which are used in the source code but are not defined.

Vocabulary An independently linked subset of the dictionary

Further information

MPE courses

MicroProcessor Engineering run the following courses:

Architectual introduction to Forth

A two day course for those with little or no experience of Forth. It provides an introduction to the architecture of a Forth system. It shows, by practical example, how software can be coded, tested and debugged, quickly and efficiently, using Forth's interactive abilities.

Embedded software for hardware engineers

A three day course for hardware engineers needing to construct real-time embedded applications using Forth cross-compilers.

Recommended reading

For an introduction to Forth:

"Starting Forth" by Leo Brodie

"Forth: A Text and Reference" by Kelly and Spies

For more experienced Forth programmers:

"Object Oriented Forth" by Dick Pountain

"Scientific Forth" by Julian Noble

Other miscellaneous Forth books:

“Forth Applications in Engineering and Industry” by John Matthews

“Stack Machines: The New Wave” by Philip J Koopman Jr

All of these books can be supplied by MPE.

Appendix A
Converting targets from v4 to v5

The main differences between v4 and v5 target source code are in the control file. Once you have installed the new system, you can modify an existing control file to match your configuration. Therefore, you must:

- define the memory map
- define how an EPROM emulator is used
- define how code is compiled into a page
- add your files into the control file

Defining the memory map

The memory map in version 4 control files are defined as in figure 20. The equivalent version 5 memory definition is shown in figure 21. The version 5 memory definition is defined by three words: **KERNEL**, **KERNEL-RAM** and **MEM-END.**

KERNEL is used to define the start and end of ROM. **KERNEL-RAM** is used to define the start and end of RAM. **MEM-END** defines the end of memory where the stacks and user areas are kept.

Using an EPROM emulator

For the version 5 compiler, you must indicate which image you want to go to the emulator. In a non-paged system, the image name is the name following the command **KERNEL**. Therefore, to send the image ROM180 to an EPROM emulator starting at address 8000h, you code:

```
$8000 IN-EMULATOR ROM180
```

This must be placed before the page is selected by **USE-CODE**.

```
\ Define the amount of RAM that can be initialised
 0400 INITIALISED-RAM                \ up to 2k bytes RAM can be set                    \ from a
table in ROM. This
                              \ equate sets the max. size
00000 ROM-BASE                      \ ROM starts at 00000                              \ will use
ORG later

\ User areas need 0100h bytes/task + 1 page for interrupts;
\ requiring 0900h bytes for a full system with 8 tasks.
\ Place INIT-U0 at the bottom of the RAM area.
\ The variable & dictionary follows, and is set by RAM-BASE
08000 EQU INIT-U0               \ task area base INIT-U0
taskram + EQU int-init-u0       \ interrupt page base
int-init-u0 intram + equ INIT-TIB       \ TIB starts at task+0900
INIT-TIB 0100 + RAM-BASE        \ Vars & Dict start at task+0A00
\ MEM-END defines the end of RAM+1
0FFFF MEM-END                   \ RAM ends at xxxx-1
```

□□□□□□ 20 □ □□□□□□□ □□□□□□□ 4 □□□□□□ □□□□□□□□□□

```
$0000 $7FFF    KERNEL ROM180          \ Define kernel ROM
$8000 $FFFF 0  KERNEL-RAM ROM180-DATA  \ Define kernel RAM
     $FFFF    MEM-END                \ End of memory
```

□□□□□□ 21 □ □□□□□□□ □□□□□□□ 5 □□□□□□ □□□□□□□□□□

Selecting the compilation page

With the version 5 cross-compiler you can generate multiple images, and code can be compiled into any image at any time. To select the page which code will be compiled into you should write:

USE-CODE <name>

where <name> is the image's name (i.e. ROM180 in the previous examples).

In a similar way, the data page may be selected:

USE-DATA <name>

where <name> is the image's data space (i.e. ROM180-DATA in the previous examples).

Appendix B
An example control file

This appendix leads you through a complete control file. It describes the use of each command followed by the usage in a typical ROM target control file. For more information on the syntax of each command see the command's glossary entry in the glossary manual.

The first page

The first page is used to introduce the rest of the file. It contains a brief (one line) description of the file's purpose followed by any other general information.

```
\ 64180 cross-compiler control file

pto
Copyright (c) 1991-93

MicroProcessor Engineering
133 Hill Lane
Southampton SO1 5AF
England

tel: (+44) 703 631441
fax: (+44) 703 339691
e-mail: mpe@cix.compulink.co.uk

First Release: 29/04/91

Changes:        See RELEASE.DOC for full details
```

Cross-compiler search order / loading macros

The cross-compiler's vocabulary search order is set so that commands can be found. You should also load any macros at this point

```
only forth definitions  decimal

all from-file nextfast                    \ load fast version of next,
```

Configuring for an EPROM emulator

The cross-compiler will download the compiled target code while it is being generated. To do this the cross-compiler needs to be told:

- the port address of the i/o card
- the type of EPROM to emulate
- the bus width of the emulated EPROM

If an emulator is not in use the following two lines should be commented out.

```
Hex 0320 Emu-Base               \ emulator i/o addr
e27256 8bit Output-Emulator      \ define EPROM & Width
```

Activating floating point

Floating point can be switched on by using the word **FLOATS**.

```
Floats                                    \ switch on floating point
```

Turning on the cross-compiler

The cross-compiler is turned on by the command **CROSS-COMPILE**. Any code compiled after this will be cross-compiled into the target image.

```
\ turn compiler on
CROSS-COMPILE
```

Setting the target search order

The target search order must be set. This tells the compiler to compile code on top of the target's Forth vocabulary.

```
only forth definitions
```

Displaying the cross-compile log

The cross-compile log can be displayed by using the word **LOG**. In this state the cross-compiler shows the type of item compiled and the target address of each item as it is compiled. This contains useful debug information but, as more text is displayed on the screen, is slower to compile. To stop the compiler from generating a full log, and just generate a dot for each definition, use **NO-LOG**.

```
no-log                                  \ no output log
```

Defining the target configuration

These flags define whether certain files are loaded, and whether certain initialisation words are included in COLD. Note that selection of the multitasker is defined by the value set for #TASKS on a later next page.

```
1 equ romforth?   \ true to load romforth extensions
0 equ paged?      \ true if target is paged
1 equ softfp?     \ true if target needs floating point
```

Defining the memory map

The memory map describes to the compiler where the start and end of ROM and RAM is. The ROM area is defined by the word **KERNEL** and the RAM area by the command **KERNEL-RAM**. The actual end of RAM is set by the command **MEM-END**.

```
\ Define memory map
  $0000 $7fff kernel ROM180
  $8000 $BFFF 0 kernel-ram KERNEL-DATA
       $C000 mem-end
```

Output into EPROM emulator

The cross compiler can send a target image directly to an EPROM emulator, which removes the time required to download the generated image. The cross-compiler needs to know what image to download (there can be several in a paged target) and where in the emulator to start downloading. The following example sets the compiler to download the image ROM180, starting at address 0000h.

```
$0000 IN-EMULATOR ROM180   \ Output to emulator
```

Selecting compilation pages

The cross compiler must be instructed into which page to compile code and data. For a non-paged system, there is only one code page and one data page, so this only needs to be done once. For a paged system, different compilation pages can be set throughout the code, so redirecting the code to different pages.

```
use-code ROM180                \ Select code page
use-data KERNEL-DATA           \ Select data page
```

Configuring for ROM PowerForth

If the ROM PowerForth utilities are being loaded, the start and end addresses of the application ROM/RAM area must be defined. For the GNC ue180 board, the application would be in the top half of memory, in RAM not yet used by the cross-compiled dictionary.

```
$C000 equ APPL-ROM                  \ where linked applications go
$FFFF equ APPL-ROM-END              \ end of linked application area
```

Defining the number of tasks

In a multitasking target the number of tasks need to be set. Each task takes up 300h bytes of RAM, so a full 8 tasks takes up 6k of RAM. If RAM usage needs to be reduced, the number of tasks can be set to the number of tasks you have.

```
$0008 Equ #tasks                \ number of tasks, at least 1
```

Defining the user area size

The user area is set by using an equate. This equate is used in a calculation before the actual user area is allocated. The user area is used to hold task specific variables such as **BASE** and **SPAN**.

```
$0100 equ User-size                 \ size of each tasks user area
```

Setting the stack sizes

The sizes of the data and return stacks must be set. These equates are used in calculations (see below) before the actual stacks are allocated in RAM.

```
$0100 equ SP-size                   \ size of each tasks data stack
$0100 equ RP-size                   \ size of each tasks return stack
```

Setting the Text Input Buffer size

The terminal input buffer is the temporary buffer that is used by the Forth interpreter.

```
$0080 equ Tib-len                    \ tib length
```

Calculating the memory per task

Each task requires its own:

- data stack
- return stack
- user area

The previous equates are used to calculate the amount of RAM required for each task. This value is set to be the equate **PER-TASK**.

```
\ Calculate memory map
User-size SP-size + Equ Task-s0     \ initial offset of data stack
Task-s0 RP-size + Equ Task-r0       \ initial offset of return stack
Task-r0  Equ per-task                        \ system area size for each task
```

Calculating the total memory requirement

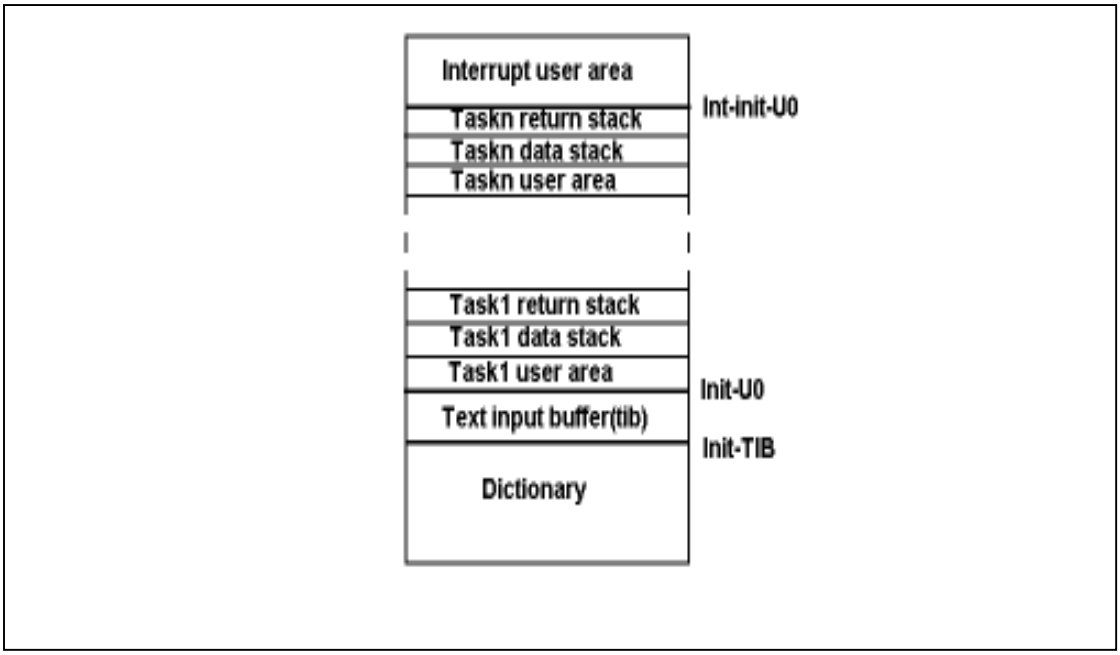
The total RAM required by the system is given by the equate **TASKRAM**. This is the amount of memory required for one task multiplied by the number of tasks in the system.

```
#tasks per-task * equ taskram        \ space used for task page
```

Setting RAM for interrupt handlers

An area of RAM is set aside for interrupt handlers exclusive use.

```
$0100 equ intram                     \ space used by interrupts
```



□□□□□□ 22 □ □□□□□□□□□□ □□ □□□

Allocation of RAM

The RAM areas for each task is allocated from the top of memory (given by EM) downwards (figure 22).

```
EM                  \ length        name
per-task -                          dup equ INT-INIT-U0              \ base of interrupt area
#tasks per-task * -                 dup equ INIT-U0 \ base of task user area
tib-len -                           dup equ INIT-TIB                 \ base of TIB
```

Setting the stack addresses

The data and return stacks are set by specifying the starting addresses for them:

```
INIT-U0 EQU INIT-R0             \ stacks inside first user area

INIT-TIB EQU INIT-S0
```

Setting the size of the task control block

The size of the task control block is set:

```
$020 EQU TASK-LEN                    \ length of task control area
```

Compiling the kernel

The main source code which makes up the interactive Forth kernel is now compiled.

```
 decimal all from-file sfr180                \ register equates
 decimal all from-file start180              \ system set up
 decimal all from-file code180               \ main code defs.
 decimal all from-file drivers\asci180       \ block and disk i/o
 decimal all from-file kernel                \ forth part 1
 decimal all from-file romtools              \ nucleus extensions
 decimal all from-file interupt              \ interrupts
```

Compiling the multitasker

In order to use the multitasker, you must compile the multitasker source code. You can use conditional compilation to force this to be compiled whenever you specify more than one task.

```
#tasks 1 >
if(
  decimal  all  from-file multi180          \ multitasker
)endif
```

Compiling the software floating point

The software floating point consists of two files, FPROMHI.FTH and SOFTFP.FTH.

```
\ Software floating point
softfp?
if(
  decimal  all  from-file softfp\fpromhi.fth          \ primitives
  decimal  all  from-file softfp\softfp.fth           \ high-level
)endif
```

Compiling the ROM PowerForth utilities

The ROM PowerForth utilities give you the ability to use host disk services from the target system.

```
\ the ROMForth files
romforth?
if(
  decimal  all  from-file romforth\link               \ linker
  decimal  all  from-file romforth\iodef              \ io definitions
  decimal  all  from-file romforth\filetran           \ source load
  decimal  all  from-file romforth\bin-down           \ binary  host
  decimal  all  from-file romforth\hex-down           \ hex  host
  decimal  all  from-file romforth\textfile           \ text files
  decimal  all  from-file romforth\blocks             \ blocks
)endif
```

Defining the target sign-on message

The target sign-on message is defined as an internal word. This makes the word unavailable for interactive use, which saves space in the target system.

```
internal
: .cpu                                  \ — ; sign on message
  ." MPE 64180 ROM PowerForth"  ;       \ sign on
external
```

Defining the last word

The last word defined is always **FORTH-83.** This indicates the end of the kernel.

```
: FORTH-83  ;     \ final word
```

Finishing the cross-compilation

The cross-compiler stops compiling when it reaches the command **FINIS.** At this point, the cross-compiler displays the cross-compile summary and prompts for a key to be pressed.

```
FINIS
```

Appendix C
Error Messages

Error messages are kept in the screen file E180.XS3 in the COMPILER directory. Error numbers start at 0, and each error number refers to a line starting at line 0.

The error messages are listed in different categaries:

- general Forth errors
- system messages
- 64180 assembler errors
- module errors
- source file errors
- DOS errors
- text file errors

General Forth Errors 0..15

These are the basic errors of a Forth system.

Error 0 - is undefined. The word is not in the dictionary search order specified, or it was misspelled.

Error 1 - empty stack, the last operation caused a stack underflow. Usually caused by using the wrong number of parameters to a word.

Error 2 - dictionary full, there is no room for more definitions. This error should not arise within the cross compiler unless you are extending it.

Error 3 - has incorrect address mode.

Error 4 - is redefined - the word's name has been used before. This is only a warning, not a proper error.

Error 5 - is undefined. See error 0

Error 7 - full stack, there are too many items on the stack. Usually caused by a stack fault in a loop.

Error 8 - cannot open **USING** file. Incorrect file name? Wrong directory?

Error 12 - uninitialised deferred word.

Error 13 - **BASE** must be DECIMAL.

Error 14 - missing decimal point. Only found when using floating point extensions.

System messages 16..31

These are error messages caused by mistreating Forth.

Error 17 - compilation only, use in definition, not when executing. Usually happens when a **;** is missing from a previous word.

Error 18 - execution only - not allowed during compilation. Usually because a **[COMPILE]** is missing in front of an immediate word.

Error 19 - conditionals not paired - overlapping control structures.

Error 20 - definition not finished - a control structure needs correction.

Error 21 - in protected dictionary - the word is below the address in **FENCE**. Not found in the cross compiler except when modifying the cross compiler, or in bizarre circumstances with Umbilical Forth.

Error 22 - use only when loading, illegal from the keyboard

Error 23 - block number out of range 0..32767 (0..7FFFh)

Error 24 - reset vocabularies - **CONTEXT** must be the same as **CURRENT** when using **FORGET**.

Error 25 - do not use when loading, only from the keyboard.

Error 26 - Initialised RAM size exceeded. Often happens when arrays are defined before variables. To reduce the size of this table, all initialised or preset RAM should be defined before arrays are used.

Error 27 - Forward references are illegal between **CREATE** ... **DOES** and **I:** ... **;** for the cross compiler.

Error 28 - word between **CREATE** ... **DOES** or **I:** ... **;** is not in host **FORTH** vocabulary

Error 29 - illegal internal value - contact MPE on (+44) 703 631441.

64180 assembler errors 32..47

Error 33 - Invalid addressing mode

Error 34 - IX or IY not allowed for this instruction

Error 35 - Bit count out of range 0..7

Error 36 - Displacement out of range -128..127

Error 37 - Invalid 16-bit register

Error 38 - Invalid interrupt mode

Error 39 - Port number out of range 0..255

Error 40 - Invalid condition code

Error 41 - Condition code error, or code not set

Error 42 - WARNING: undocumented but regular instruction

Error 44 - Invalid source addressing mode

Error 45 - Invalid destination addressing mode

Error 46 - Unconsumed reference. This error is usually caused when op-codes or addressing mode indicators like **(IX)** are misspelled. The mis-spelled word is then seen as a forward reference which has not been used by any opcode. This check is performed by **END-CODE** or **FORTH** and thus will not be seen until the end of a section of code.

Error 47 - Code error, stack depth changed. This is general catch all error performed by **END-CODE**.

Module errors 48..63

Error 49 - public words table full - max 32 (decimal) words/module

Error 50 - module number out of range 0..31 (decimal)

Error 51 - slot already occupied - slot must be empty before entry is made

Error 52 - not enough memory - fit more! - RAM is cheap!

Error 53 - can't load module file - DOS can't find it, or can't read it

Error 54 - can't free memory - DOS won't let go - see DOS function 49H

Error 55 - module not present - requested module is not resident

Error 56 - external references table full - max 32 (decimal) words/module

Error 57 - unresolved external reference - use RESOLVE-ALL before execution

Error 62 - illegal operation in slave module

Error 63 - illegal operation in master module

Source file errors 64..79

These errors are given by the screen file handlers.

Error 65 - no screen file open. Often a result of a previous operation failing to open or reopen a file.

Error 66 - screen file seek error.

Error 67 - screen file write error.

Error 68 - path not found. Usually because the file or path name has been misspelled.

Error 69 - starting screen number less than ending screen number.

DOS errors 80..112

Error 81 - invalid function number - DOS doesn't know what to do

Error 82 - file not found - wrong directory or doesn't exist

Error 83 - path not found - incorrect spelling? - device not installed?

Error 84 - no handle available - all handles are in use

Error 85 - access denied - e.g. attempt to write to read-only file

Error 86 - invalid handle - file/path not open?

Error 87 - memory control blocks destroyed - whoops!

Error 88 - insufficient memory - 640k/1Mb is not enough

Error 89 - invalid memory block address - DOS did not allocate this segment

Error 90 - invalid environment - previous SET or PATH command bad

Error 91 - invalid format - ask Microsoft what this one means

Error 92 - invalid access code

Error 93 - invalid data

Error 95 - invalid drive specification

Error 96 - attempt to remove current directory

Error 97 - not same device

Error 98 - no more files to be found

Text file errors 112..127

These errors are issued by the text file handler.

Error 113 - cannot allocate memory. Each nested file needs about 9k bytes.

Error 114 - cannot free memory. Usually a symptom of something running amok.

Error 115 - cannot open file. Usually because of a misspelled name.

Error 116 - cannot close file. Usually a symptom of something running amok.

Error 117 - cannot seek to byte requested in file. Usually a symptom of something running amok.

Error 118 - read-path error. Disk cannot be read, normally seen only from floppy disks, or failing hard discs.

Error 119 - file nesting depth reached - cannot open another file. You have nested files too deep.

Error 120 - file de-nesting error. Usually a symptom of something running amok.

Error 121 - start page number greater than last page number in file.

Appendix D
Technical support

Technical Support

Technical support is available from MPE during office hours (9am-5pm), or via access to our conference on the CIX (Compulink Information eXchange) bulletin board system. MPE has its own technical support conference on the CIX bulletin board system. You can also obtain technical support via email.

Before calling MPE make sure you have the following by the telephone:

- The serial number of your software (it is on the original disks)
- Your compiler manuals
- A computer, with the software running on it
- Any other relevant information

You should also know the type of processor and the amount of RAM your machine has installed.

A fax describing the problem, sent before you telephone is often the quickest way of getting an answer. This is because many problems which at first glance seem complex often have a very simple solution and an application note may be available.

MPE Tel: +44 703 631441
MPE Fax: +44 703 339691

CIX (voice): +44 81 390 8446
CIX (data): +44 81 399 1244

Internet: mpe@cix.compulink.co.uk

Blank page

Index

F

H

I

K

L

M

U

V

X