MPE Forth 5 for RTX20xx

User Manual

MPE Forth 5 for RTX2000

USER MANUAL

RTX2000/1A/10 Target

Version: 5.100

User Manual

Revision: 1.02

Date: 17 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
Email: mpe@cix.compulink.co.uk

MPE Forth 5 for RTX2000

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

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.

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

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 RTX2000**

Processor and computer type:

Operating system:

Disc format: (3½, 5¼)

Serial number:

Purchase date:

Bought from:

Table of Contents

Blank Page

List of Figures

List of Tables

Blank Page

Installing the system

It is recommended that you install the MPE Forth 5 RTX2000 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. Drives A: and B: are included for installing onto a network.

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 choose 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
- you have recently upgraded
- 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.

Recent upgrade

If upgrading your development system, select *standard*. This installs the whole system as software versions are 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 the *don't install a driver* .

The emulator base address

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

PowerForth Plus path

PC PowerForth Plus is a Forth for your PC. Type the path of where you want it to be installed. Press return 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 return 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 place where XShell and PC PowerForth Plus 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.

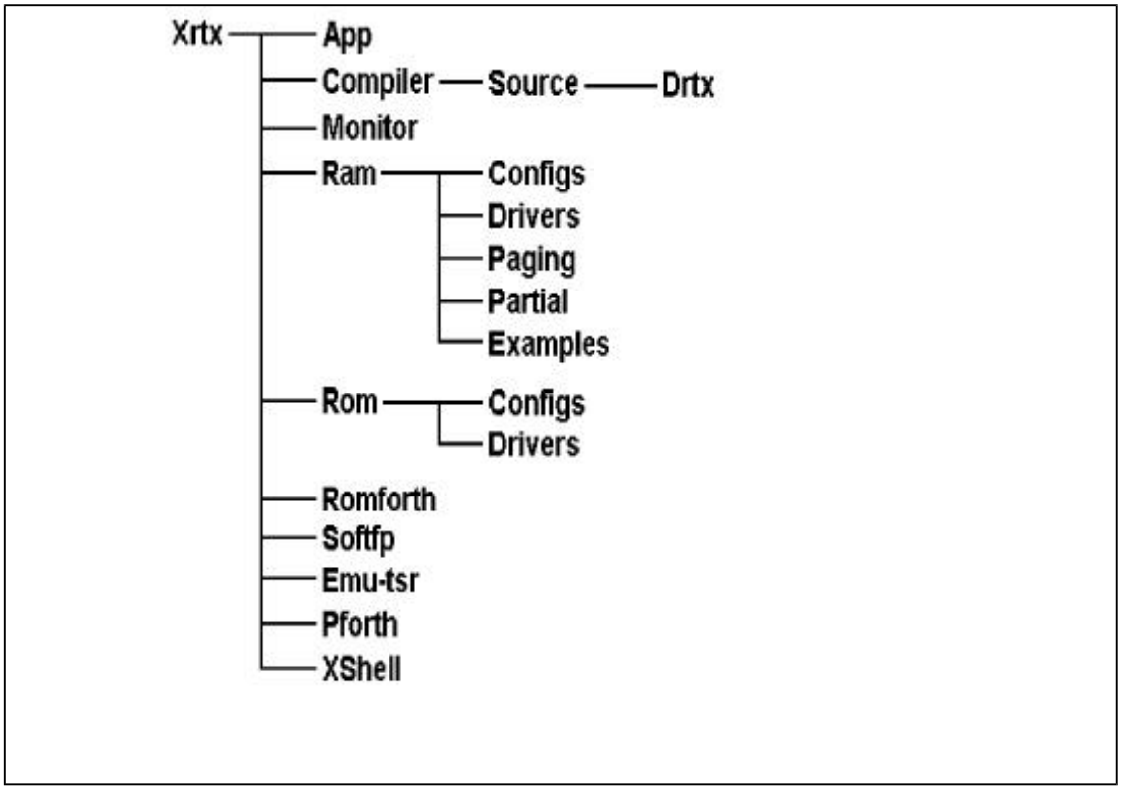

Figure 1 - The development system's directory structure

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 directory COMPILER\SOURCE and COMPILER\SOURCE\DRTX.

ROM and RAM target Forth

Source code is supplied for developing ROM and RAM target Forths. 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 either a ROM or RAM target Forth. The installer places the ROM target source code in the

directory ROM, and the RAM target source code in the directory RAM. See chapters 3 and 4 on how to generate ROM and RAM target Forths.

Umbilical Forth

Source code is supplied to generate an Umbilical Forth. Umbilical Forth is a significantly smaller Forth than the ROM target Forth, so an interactive Forth can be generated which is smaller than 4K. 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 directories APP and MONITOR.

Leburg EPROM emulator drivers

The cross compiler can directly download code, as it is generated, to a Le-Burg 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 generating 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 an MPE RTX board you can use the supplied control files. There are files for 2000, 2001 and 2010 variants of the RTX processor. By using one of these the installation of a ROM target Forth for your board will be greatly simplified. The control file to use will depend on the type of processor and the clock crystal fitted to your board. These files are in the directory ROM\CONFIGS.

If you do not have an MPE board you will have to modify a control file and serial line drivers to suit your own board.

The control file

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

- 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.

Setting the code generator

The code generator will by default generate code for a RAM Target. If you wish to produce code for a ROM target you will need to inform the compiler. This may be done by using the **ROM-TARGET** directive directly after **CROSS-COMPILE**. Thus you would code:

```
Cross-Compile                   \ generate target code
Rom-Target              \ for ROM target
```

The memory map

The memory map describes the addresses where ROM and RAM start and end in your target system. The memory map is described to the cross compiler in your control file.

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 two parts:

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

Setting the start and end of ROM

The start and end of ROM is 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 12, Paged targets.

Setting the start and end of RAM

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

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

where ram-start is the address of the start of RAM, ram-end is the address of the end of RAM, page-id is a unique identifier for this area of memory and <name> is the name for this area of memory. The numbers ram-start and ram-end are, by default, in decimal, but can be entered in hex by preceding then by a $.

The label <name> is the name of the kernel's data area in a paged system. In a non-paged system <name> is not actually used but must be stated. In a non-paged system, page-id can be set to any number. For more information on paged systems, see chapter 12, Paged targets.

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**.

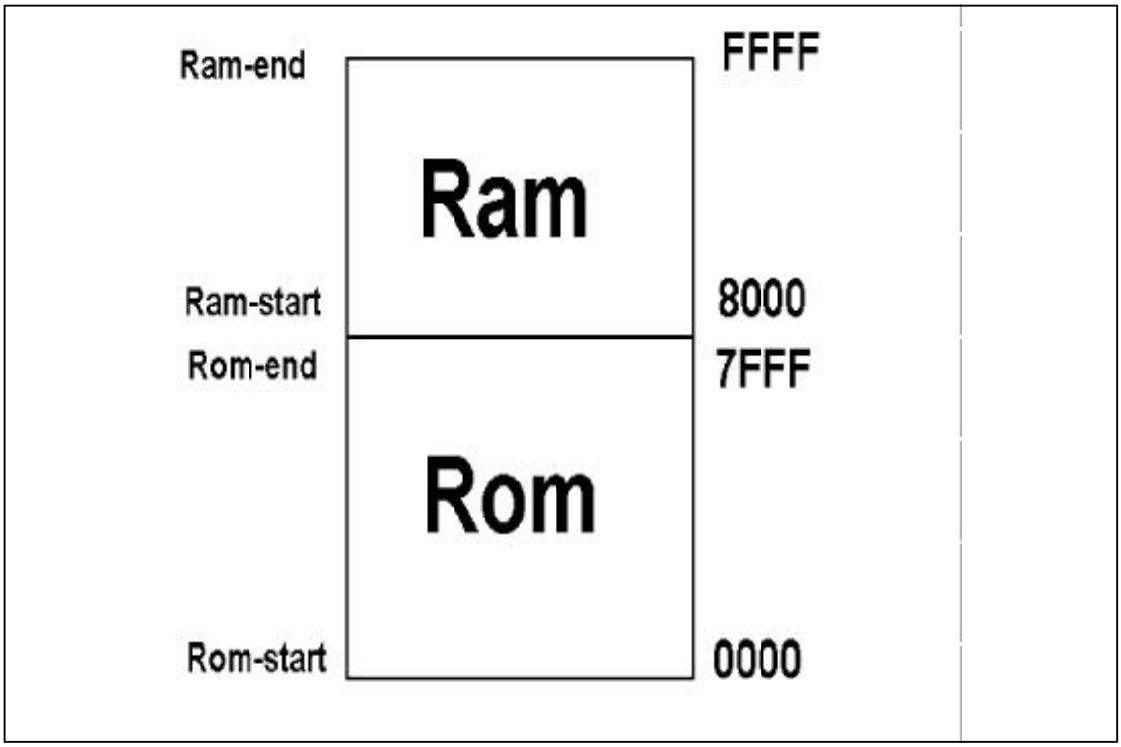

Figure 2 - An example memory map

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 Kern
$8000 $FFFF  1 KERNEL-RAM Kern-data

USE-CODE Kern
USE-DATA Kern-data
```

This indicates two areas of memory with names Kern and Kern-data.

Modifying the serial line drivers

Your target board communicates with the the external world via a UART Unlike some other processors, the RTX does not have an onboard UART. However, if you are using an 8530 serial communications device, the supplied serial driver code can be used. This is in the directory ROM\DRIVERS.

If you are using a different UART 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

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 which makes a more responsive target. The RTX will normally function correctly at 38400 baud.

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
- increment 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 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, XRTX\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, i.e ALL FROM-FILE CONFIGS\CONTROL.CTL followed by ENTER

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 has been setup, you can 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 compiled type of item is coded as two characters as in table 1.

Turning on and off the log

Instead of having the data displayed for each compiled item, you can chose to only display a dot. The advantage of this is that the compiler spends less time displaying data and so the cross compile 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.

Code	Compiled type	Code	Compiled type
VR	Variable	FV	Floating point variable
CN	Constant	FC	Floating point constant
LB	Label	FA	Floating point array
:	Colon defintion	EQ	Equate
CD	Code definition	CR	Create ... Does>
DF	Deferred word	US	User variable
VC	Vocabulary		

Table 2 - Key to cross-compiler log

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 spelling mistakes or some of the code is not being compiled.

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 compilation 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 setup 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 setup 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 chapter on how to setup 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 setup already. To run the programmer's software press F6. To setup XShell to use a EPROM programmer, refer to the XShell chapter.

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.

Resetting the target board

```
MPE RTX 2000 ROM PowerForth v3.00
27384 bytes free

 ok
```

Figure 3 - The target sign-on

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 now should 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**.

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 generate ‘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**, this 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.

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 chapter.

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(press F4 or F6)
- reset the target

The target board should now sign-on, and you can test your application.

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

 MAKE-TURNKEY MY-APP
```

Figure 4 - Example turnkey 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**.

Blank Page

Generating a RAM target Forth

This chapter describes how to generate a RAM 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 generating 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.

What is a RAM target?

A RAM target has both EPROM and RAM. However, on power-up the entire contents of the EPROM are copied into RAM and execution takes place from there.

Is your board already supported?

If you have an MPE RTX board you can use the supplied control files. There are files for 2000, 2001 and 2010 variants of the RTX processor. By using one of these the installation of a ROM target Forth for your board will be greatly simplified. The control file to use will depend on the type of board you have. If you have an STE variant you should use one of the STExxxx.CTL files as your control file. If you have a PowerBoard you should use a file xxxxPBxx. The exact file you choose will depend on the processor and the clock crystal fitted to your board. These files are in the directory RAM\CONFIGS.

If you do not have an MPE board you will have to modify a control file and serial line drivers to suit your own board.

The control file

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

- 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 RAM\CONFIGS.

The memory map

The main difference between a RAM target and a ROM target is that all code and data reside in RAM during execution. All code and data is copied from ROM into RAM at startup, and then executed from RAM. From the target's point of view there is no distinction between ROM and RAM. However you still need to declare to the compiler where the start and end of your memory area is. The memory map is described to the cross compiler in your control file.

The memory map is defined by the:

- start of RAM
- 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 two parts:

- the start and end of RAM
- a dummy page for use by the cross compiler

Setting the start and end of RAM

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

ram-start ram-end KERNEL <name>

where ram-start is the address of the start of RAM, ram-end is the address of the end of RAM 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 ram-start and ram-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 12, Paged Targets.

Setting the dummy page

In a ROM target, code and data are split into two memory areas, one for code and one for data (see chapter 3 for more details)The start and end of the data area is defined by using the compiler directive **KERNEL-RAM**. In a RAM target, this distinction does not exist, but the compiler still needs a memory area to store its internal labels for variable values and the like. You must de-

clare a dummy page so the compiler will allocate memory correctly. This is done in the form:

```
0 0 page-id DATA-PAGE dummy
```

where the two zeros are simply arbitrary numbers, page-id is a unique identifier (one which you will not be using elsewhere) and dummy is simply an arbitrary label, which is not actually used, but must be stated.

Since there is no **ROM-TARGET** directive, the compiler will assume that you are using a RAM target, and generate code accordingly.

Setting the compilation pages

In a non-paged system, the compiler must be instructed to compile into the page defined by **KERNEL.**

```
USE-CODE <name1>
USE-DATA <dummy>
```

where <name1> is the name of the kernel page defined with **KERNEL** and <dummy> is the compiler's dummy page defined with **DATA-PAGE.**

An example

If your target board has a typical memory map, with RAM from 0000h to FFFFh, your control file should be modified so that it reads,

```
$0000 $FFFF KERNEL Kern
$0000 $0000  0 DATA-PAGE Dummy

USE-CODE Kern
USE-DATA Dummy
```

This indicates a single area of memory with the name Kern.

Modifying the serial line drivers

Your target board communicates with the the external world via a UART Unlike some other processors, the RTX does not have an onboard UART. However, If you are using an 8530 serial communications device, the supplied serial driver code can be used. This is in the directory RAM\DRIVERS.

If you are using a different UART 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 RAM\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 which makes a more responsive target. The RTX will normally function correctly at 38400 baud.

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
- increment 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 deferred 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 deferred word **KEY**. The stack effect of **(KEY)** is:

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

Detecting a received character

The target needs 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, XRTX\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 RAM 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, i.e ALL FROM-FILE CONFIGS\CONTROL.CTL followed by ENTER

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 RAM 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 has been setup, 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 mes-

Code	Compiled type	Code	Compiled type
VR	Variable	FV	Floating point variable
CN	Constant	FC	Floating point constant
LB	Label	FA	Floating point array
:	Colon defintion	EQ	Equate
CD	Code definition	CR	Create ... Does>
DF	Deferred word	US	User variable
VC	Vocabulary		

Table 3 - Key to cross-compiler log

sage 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 short-

ened name. The compiled type of item is coded as two characters as in table 3.

Turning on and off the log

Instead of having the data displayed for each compiled item, you can chose to only display a dot. The advantage of this is that the compiler spends less time displaying data and so the cross compile 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

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

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 compilation 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 setup 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 setup 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 chapter on how to setup 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 setup already. To run the programmer's software press F6. To setup XShell

```
MPE RTX 2000 RAM PowerForth v3.00
Copyright (C) 1988, 1989, 1990 Microprocessor Engineering
Free Dictionary Space: 29556 bytes
  ok
```

Figure 5 - The target sign-on

to use a EPROM programmer, refer to the XShell chapter.

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.

Resetting 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 5. The version number and the number of bytes free will

depend on your system. You now should 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 without actually running any Forth.

The memory map definition.

If the memory map for either the ROM or RAM definitions is wrong. The target may not sign-on at all or may generate '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**, this 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.

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 chapter.

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.

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

 MAKE-TURNKEY MY-APP
```

Figure 6 - Example turnkey application

Running your application

To compile the application you need to:

- run the cross compile(press F3)
- download to the EPROM emulator/programmer(press 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**.

Blank Page

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, the generating 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 MPE RTX Powerboard you can use the supplied control files. There are files for the 2000 (8 and 10 MHz), 2001A and 2010 variants of the RTX processor. A control file is also supplied for use with the Harris RTX DB board. By using one of these, the installation of an Umbilical Forth for your board will be greatly simplified. The control file to use will depend on the type of board you have. The supplied control files are in the directory MONITOR\CONFIGS.

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

The Umbilical system

An RTX umbilical system is generated in two parts, the monitor and the application. The monitor has three functions:

- to download an application
- to execute application code
- to provide I/O services to the application

In order to run an application you must have a working monitor installed on your board, in EPROM or an EPROM emulator. You can then cross compile the application and download it via the serial line.

The monitor control file

The monitor control file contains all the details of your board that the cross compiler needs to know. This includes:

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

As well as containing configuration information, the control file contains a list of files which are to be cross compiled. These files are used to generate the monitor for the board.

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 MONITOR\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 described to the cross compiler in your control file.

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 two parts in your control file.

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

Setting the start and end of ROM

The start and end of ROM is 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 cross compiler automatically adds the extension .IMG to **<NAME>** when saving the file. The numbers rom-start and rom-end are, by default, in decimal, but can be entered in hex by preceding them by a $.

<NAME> is also the name of the kernel page in a paged system. For more information see Paged targets, chapter 12.

Setting the start and end of RAM

The start and end of RAM is defined by using the compiler directive **DATA-PAGE**. **DATA-PAGE** is used in the form:

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

where ram-start is the address of the start of RAM, ram-end is the address of the end of RAM, page-id is a unique identifier for this area of memory and **<NAME>** is the name for this area of memory. The numbers ram-start and

ram-end are, by default, in decimal, but can be entered in hex by preceding the value with a $.

The label **<NAME>** is the name of the kernel's data area in a paged system. In a non-paged system **<NAME>** is not actually used but must be stated. In a non-paged system, page-id can be set to any number. For more information on paged systems, see the chapter on paged targets .

Setting the compilation pages

In a non-paged system, the compiler must be instructed to compile into the pages defined by **KERNEL** and **DATA-PAGE**. 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 RAM page defined with **DATA-PAGE.**

An example

For example, if your target board has a memory map as in figure 2, your con-

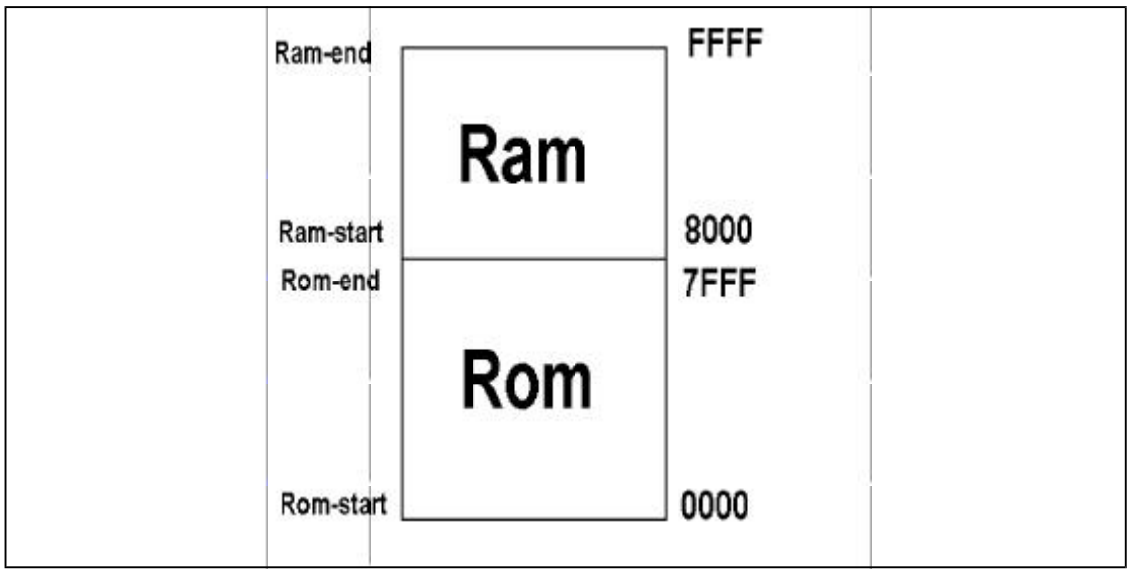

Figure 7 - Example memory map

trol file should be modified so that it reads,

$0000 $7FFF KERNEL Kern
$8000 $FFFF 0 DATA-PAGE Kern-data

```
USE-CODE Kern
USE-DATA Kern-data
```

This indicates two areas of memory (pages) with names Kern and Kern-data

Modifying the serial line drivers

Your target board communicates with the the external world via a UART. If you are using an 8530 or 2691 UART, one of the supplied serial driver files can be used. These are in the directory MONITOR\DRIVERS.

If you are using a different UART 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

As with the control file it is normally easier to modify an existing serial line driver file rather than creating your own from scratch.

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 initially set to 2400 baud until the target board is working. It can then be raised to 9600 or above which makes the target seem more responsive. The RTX will normally run without problems at 38400 baud.

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

Running XShell

If you allowed the installer to modify your AUTOEXEC.BAT, you just need to type XS3 to run XShell. If you didn't, then you need to state the full path of XShell. The installer will place XShell in the directory, XRTX\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 MONITOR 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, i.e ALL FROM-FILE CONFIGS\CONTROL.CTL followed by ENTER

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:

i) run Xshell while in the MONITOR 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 the monitor

Now the hardware and software has been setup, you can now cross compile the monitor source code. This should then be either downloaded into an EPROM emulator or blown into EPROM.

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 1.

Turning on and off the log

Instead of having the data displayed for each compiled item, you can chose to only display a dot. The advantage of this is that the compiler spends less time displaying data and so the cross compile is quicker. To do this, change

Code	Compiled type	Code	Compiled type
VR	Variable	FV	Floating point variable
CN	Constant	FC	Floating point constant
LB	Label	FA	Floating point array
:	Colon defintion	EQ	Equate
CD	Code definition	CR	Create ... Does>
DF	Deferred word	US	User variable
VC	Vocabulary		

Table 5 - Key to cross-compiler log

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 number of forward references made

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.

The forward reference count simply informs you of the number of words defined after they are first used.

Problems, Problems ...

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

Cross compiling the Forth kernel

Once you have a monitor you can cross compile the Forth kernel code and download it to the target. There is a control file for the kernel code, which is laid out in a similar fashion to the monitor control file. Example kernel control files are in the directory APP\CONFIGS.

The main difference between the two is that the kernel control file ends with the **UMBILICAL-FORTH** directive. This instructs the cross-compiler to run an umbilical Forth system. Once this is complete you will be instructed to reset your target board, and download the kernel.

Downloading the Forth kernel

Once you have reset your board you will be asked whether you wish to "download files to target?" Whilst developing you should answer "Y" as you will not have your Forth kernel and application in EPROM. It is only at the final test stage when the application code is in EPROM that you will not need to download it. During the download stage, a dot will be produced for

```
Please reset target system, and then
press <space> to run Umbilical Forth.


Download files to target? (y/n) Y


Downloading APP2000.IMG to page 00, offset 8100  ...
```

Figure 8 - Umbilical download messages

```
Umbilical Forth v3.00
Target: Harris RTX 2000/20001A/2010
Copyright(C) 1990,91 Microprocessor Eng. Ltd.

BASE now in DECIMAL

 ok
```

Figure 9 - The Umbilical Forth sign-on

each 1K block of code that is being downloaded. An example screen display is shown in figure 8

Running the target Forth

Once the image generated by the compiler has been downloaded to the target, it is ready to be tested.

The sign-on

You will see a message similar to that in figure 9. 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 the 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.

Assuming all is well you can proceed to generate 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 chapter.

Modifying the control file

Once your application has been written, you can add it to the kernel 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 should add them to the end of the list.

Running your application

To compile the application you need to:

- run the cross compile(press F3)
- reset the target
- download the application to the target board

The target board should then sign-on. You can now test the 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>

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

MAKE-TURNKEY MY-APP
```

Figure 10 - Example umbilical turnkey application

where **<name>** is the name of the word to run at startup. The word **<name>** must be defined before using this directive. If your application uses the serial line you will also need to ensure that you are using a driver which does not use the umbilical mechanism. The example in figure 10 generates a simple turnkey application when cross compiled. To see the example working, you must switch XShell into target mode.

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 the 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. Normally, your application has grown too large for your ROM space. You can reduce the size of your compiled image 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.

Factorising 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 2 more bytes than a constant, every time it is refered 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.

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 4k). 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 RTX processors have a high degree of paralellism in their architecture. It is sometimes possible to merge several instructions, and hence increase the execution speed. See the chapter on Assember Opcodes to find out how to do this.

Blank Page

Assembler Opcodes

Introduction

In most conventional Forth implementations, one Forth primitive corresponds to several machine instructions. In the case of the RTX family of processors, several Forth primitives can be combined into one instruction. There is therefore no assembler required for creating Forth primitives. However, it is possible to combine several Forth primitive instructions into one RTX instruction. Doing this, allows those primitives to be executed in the same clock cycle, and occupy one memory cell. How to combine primitive instructions is described in the section below about building new opcodes.

For a full description of the processor instruction set, you should consult the excellent 'Harris RTX-2000 Programmer's Reference Manual', supplied separately. This volume contains a description of all the useful instructions, and provides a valuable insight into programming the RTX processors.

Processor Architecture

The RTX-2000 is a two stack processor. One stack is used to hold data to be acted on by primitive instructions and secondary subroutine calls, whilst the other is used to store subroutine return addresses and other housekeeping values. Each instruction on the RTX-2000 is 16 bits wide. There are on-chip registers in the processor core, each also 16 bits wide. These hold the stack pointers, act as temporary storage during arithmetic calculations and map I/O.

There is no decode ROM in the processor to protect it from errant instuctions. For most instructions, all bits are significant, and each controls hardwired logic in the processor core.

Building New Opcodes

The user is provided with a named set of instructions which match the Forth word set, but other instructions can be formed as required, and the facilities for generating new instruction mnemonics are described below.

Why should you build a new opcode?

The highly parallel nature of the processor, gained by using many internal buses, means that two consecutive instructions can occur that use independent parts of the processor. If this happens, it is usually true that the different parts of the processor are controlled by separate sets of bits in the processor opcode. The two instructions can be merged to form a new instruction, halving the time taken for the sequence to execute. This merging procedure is referred to as optimisation.

To avoid generating many mnemonics to describe these merged opcodes, the compiler contains a set of rules for merging opcodes automatically. A list of predefined opcodes is given later.

When to build a new opcode

If an opcode is used only once it is not necessary to build a special opcode for it and the code can be inserted ‘in-line’. However, if an opcode will be used many times it is often worth making a special word for it. This sort of decision is similar to that taken when factorising normal Forth code.

N.B. When building new opcodes be careful to test them interactively first. Since the processor has no protection against illegal instructions, and most instructions will do something, the result of a mistake is often having to reset the processor.

How to define an opcode in-line

An opcode can be inserted into a definition, using the **[** and **]** operators which temporarily turn off the compiler. This allows you to put the instruction directly into the dictionary using Forth’s ‘**,**‘ word. For example, a computed goto can be generated by writing into the program counter with the return bit set. The following sequence compiles this opcode:

```
..... [ $BEA7 , ] .....
```

How to define a new opcode permanently

You can define a new opcode by using the defining word **UCODE** in the following way:

nnnn UCODE <name>

This will automatically generate code to lay down the required opcode when its name is specified. This is similar to the way a normal Forth word (defined using a colon) will generate code to execute itself when its name is specified.

For example the RTX-2000 opcodes **SELDPR** (fetch/stores in data page) and **SELCPR** (fetch/stores in code page) can be added as follows.

```
$B08D UCODE SELDPR          \ define opcode
$B00D UCODE SELCPR          \ define opcode
```

You can now use **SELDPR** and **SELCPR** in the same way as you would use any other Forth word such as **DROP** or **2+**.

How to Control the Optimiser

In some cases it may be necessary to inhibit the optimiser's action, or modify it in some way.

Why should you modify the optimiser's action?

When the opcode involves the return stack it is often necessary to turn off the optimiser. Another common occasion is with the word **COMPILE** which later compiles the word after itself. If the next two words can be merged, the action that will be compiled will be the compound action, which is not what is usually required. In this case it is necessary to turn off the optimiser so that the optimisation does not happen.

How to modify the optimiser's action

Two opcodes can be kept separate by using the immediate word **[\\]**. For example, the sequence

```
COMPILE +  2*
```

would later compile the merged opcode

```
+ 2*
```

whereas

```
COMPILE +  [\\] 2*
```

breaks the optimisation leading to the desired result. There is a non-immediate form \\ which is most often used inside defining words or structure control words to prevent undesired merging when the opcode or structure is compiled.

This feature of using the optimiser to pile up opcodes can be used to good effect with the streaming instruction **TIMES** (also referred to as **OF(** in the Harris documentation) to build up a compound instruction that is repeated.

Predefined Opcodes

A number of opcodes are predefined in the cross compiler. These can be used just like Forth words, but because they are defined as RTX opcodes, they execute in one or two cycles only. For convenience, they are presented as tables of related instructions. However, for detailed information about the RTX opcode set, you should consult the Harris documentation.

Opcode	Name	Opcode	Name
0A096	RTR	0A09E	RDR
0A196	R’	0AADE	C’
0B010	-SOFTINT	0B090	SOFTINT
0B00D	SELCPR	0B08D	SELDPR

Table 6 - Special register opcodes

Opcode	Name	Opcode	Name
0BE03	CR@	0BE83	CR!
0BE04	MD@	0BE84	MD!
0BE05	SQ@	0BE85	SQ!
0BE06	SR@	0BE86	SR!
0BE07	PC@		
0BE08	IMR@	0BE88	IMR!
0BE09	SPR@	0BE89	SPR!
0BE0B	IVR@	0BE8B	SLR!
0BE0C	IPR@	0BE8C	IPR!
0BE0D	DPR@	0BE8D	DPR!
0BE0E	UPR@	0BE8E	UPR!
0BE0F	CPR@	0BE8F	CPR!
0BE10	IBC@	0BE90	IBC!
0BE11	UBR@	0BE91	UBR!
0BE13	TC0@	0BE93	TC0!
0BE14	TC1@	0BE94	TC1!
0BE15	TC2@	0BE95	TC2!

Table 7 - Opcodes for internal register access

Opcode	Name	Opcode	Name
0A41A	U/1’	0A45A	U/’
0A458	U/"	0A51A	S1’
0A55A	S’	0A558	S"
0A49C	U*"	0A89C	U*’
0A89D	*’	0A49D	*"

Table 8 - Opcodes for step mathematics

Opcode	Name	Opcode	Name
0BE00	G@	0BE80	G!
0CE00	U@	0CE80	U!
0EE00	@	0EE80	!
0FE00	C@	0FE80	C!
0E942	@++	0E9C2	!++
0F941	C@++	0F9C1	C!++
0E940	@+	0E9C0	!+
0E540	@-	0E5C0	!-
0F940	C@+	0F9C0	C!+
0F540	C@-	0F5C0	C!-

Table 9 - Memory access opcodes

Opcode	Name	Opcode	Name
		0A001	0
0A002	2*	0A003	2*C
0A004	CU2/	0A005	C2/
0A006	U2/	0A007	2/
0A008	N2*	0A009	N2*C
0A00A	D2*	0A00B	D2*C
0A00C	CUD2/	0A00D	CD2/
0A00E	UD2/	0A00F	D2/

Table 10 - Shift opcodes

Opcode	Name	Opcode	Name
0A040	NIP	0AE40	DROP
0A0C0	DUP	0AEC0	OVER
0AE80	SWAP	0BE00	R@
0BE01	R>	0BE81	>R

Table 11 - Stack operator opcodes

Opcode	Name	Opcode	Name
0A240	AND	0A340	NOR
0A440	SWAP-	0A540	SWAP-C
0A640	OR	0A740	NAND
0A840	++	0A940	+c
0AA40	XOR	0AB40	XNOR
0AC40	-	0AD40	-C

Table 12 - Dyadic ALU opcodes

Optimiser Glossary

The following words may be used to control the way in which the compiler optimises the code.

Opcode	Name	Opcode	Name
0A000	NOOP	0A000	NOP
0B8C0	0+	0B8C1	1+
0B8C2	2+	0B4C1	1-
0B4C2	2-	0A012	2*’
0BCC0	NEGATE	0DE00	LIT
0BE82	TIMES	0BE82	Of(
0BE82	EXTRA(	0BE87	EXECUTE
0BEA7	GOTO	0BEA0	;R
08800	?BRANCH	09000	BRANCH
09800	(NEXT)	0A100	NOT

Table 13 - Miscellaneous opcodes

[\\] —
“bracket-stop-opt”

Used during compilation to break the optimisation sequence at this point. Opcodes either side of **[\\]** will not be merged. See **NO-OPTIMIZE OPTIMIZE**

NO-OPTIMIZE —
“no-optimise”

This directive disables the opcode optimiser. The optimiser can merge several opcodes into one, so producing smaller and faster code. The default state is **OPTIMIZE**. The usual reason to turn the optimiser off is in order to build a table of opcodes.
For example:

```
CREATE OPCODE-TABLE
 ] @ ! + 0< - R> DROP >R R@ [
```

Without any optimisation control, several elements of this table would be merged, producing an incorrect result. The optimiser can be 'broken' using the **[\\]** function as follows:

```
CREATE OPCODE-TABLE

 ]  @ [\\] ! [\\] + [\\] 0< [\\] - [\\] R> [\\]
 DROP [\\] >R [\\] R@ [\\] [
```

The use of **[\\]** after each opcode forces the optimiser not to merge any opcodes, but makes the code longer and difficult to read or understand. A better solution is to use **NO-OPTIMIZE** and **OPTIMIZE** to turn the optimiser off for a short while.

```
NO-OPTIMIZE

CREATE OPCODE-TABLE
 ]  @ ! + 0 - R DROP R R@  [

OPTIMIZE
```

NO-TAIL-OPT —

"no-tail-opt"

A procedure that ends in a call to another procedure, followed by a return can often by optimised by just branching to the other procedure. Because some Forth words can make assumptions about the use of the return stack, this optimisation is controlled separately from all other optimisations. The default is **NO-TAIL-OPT**. It is recommended that you develop the code without this optimisation, and then turn it on for final debugging. **NO-TAIL-OPT** and **TAIL-OPT** can be used around single words if required.

```
: A  ..... ;
: B  ..... A ;
```

If **TAIL-OPT** is active the last call to **A** will be converted to a jump, saving two bytes and one clock cycle. If this is undesirable use the code below.

```
: A  ..... ;

NO-TAIL-OPT

: B  ..... A ;

TAIL-OPT
```

OPTIMIZE —
"optimise"

This directive enables the opcode optimiser. For details see **NO-OPTIMIZE**.

TAIL-OPT —
"tail-opt"

Turns on branch optimisation. The default is OFF. See **NO-TAIL-OPT**

Multitasker

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

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

The multitasker is in the file MULTIRTX.FTH in the \ROM and \RAM directories.

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. How to do this at run-time is dealt with later.

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, by default, 4 but can be set to a lower number. This reduces the amount of memory that is allocated to all the tasks., so leaving more RAM for your application.

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

Figure 11 - Multitasking example

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 its 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.

An example

An example task is shown in figure 11. 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 11, 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 being 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 cycles to stop for.

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. 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

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 14 - Multitasker data structure

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.

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

Table 15 - A task's status word

Receiving a message

To receive a message, use **RECEIVE-MESSAGE**. **RECEIVE-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.

Setting a bit in the status word.

A bit can be set in a tasks 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 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 an hash (#) every so often, but leaving the forground Forth running. To use the multitasker you must cross-compile the file MULTIRTX.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 couple of 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 way:

- 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 interrupt and 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"

Returns the task number of the currently executing task (oneself).

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 are received.

Interrupts

This chapter describes how to write interrupt handlers in Forth. It details how to setup and control interrupt handlers.

Interrupts on the RTX2000 Family

When an interrupt occurs on an RTX processor, execution starts from a location in the vector table. Each interrupt source is allocated a 32 byte block in the table, which can contain a small service routine or any other code that the user wishes to place there.

The location of the vector table is user selectable by writing to bits 10-15 of the IBC register. In order that the supplied handlers can be used, the equate **INT-BASE** should be set in the control file. The vector table MUST be aligned on a 1Kbyte boundary.

There are a number of possible interrupt sources on an RTX processor. These are shown in table 16. For more information on the interrupts for your processor, refer to your 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.

Setting an interrupt

An interrupt is set by using the deferred words in table 16. For example, if you wish to run your word UNDERFLOW-ERR if the parameter stack un-

Offset into Vector Table	Deferred word	Source
0000h		Reserved
0020h		Reserved
0040h	swi-isr	Software interrupt
0060h	ei5-isr	External interrupt 5
0080h	ei4-isr	External interrupt 4
00A0h	ei3-isr	External interrupt 3
00C0h	timer2-isr	Timer/Counter 2
00E0h	timer1-isr	Timer/Counter 1
0100h	timer0-isr	Timer/Counter 0
0120h	ei2-isr	External interrupt 2
0140h	rsv-isr	Return stack overflow
0160h	psv-isr	Parameter stack overflow
0180h	rsu-isr	Return stack underflow
01A0h	psu-isr	Parameter stack underflow
01C0h	ei1-isr	External Interrupt 1
01E0h	nmi-isr	NMI
0200h	phantom-isr	No interrupt

Table 16 - RTX vector table

derflows in your code, you need to assign an action to the deferred word PSU-ISR (table 16). Therefore your source code should read:

```
ASSIGN UNDERFLOW-ERR TO-DO PSU-ISR
```

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 setup or the interrupt handler assigned to the deferred word before the interrupts are enabled. You will also need to unmask the individual interrupt control bit for the interrupt you are interested in.

Special Note

There is a bug in the NMI interrupt on the RTX2000 and 2001 processors, which means it is not safe to perform a return from this type of interrupt. The NMI should therefore be used **only** when a return will not happen. This bug has been corrected on the 2010 with the addition of an extra NMI mode.

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.

Unmasking an individual interrupt

The RTX processors allow you to selectively enable and disable individual interrupts by use of an interrupt mask register. To enable a specific interrupt, specify the name of the interrupt followed by the word **UNMASK**. Note that you must still enable interrupts globally using **EI**.

Masking an individual interrupt

An individual interrupt may be disabled by specifying its name followed by the word **MASK**. Once this has been executed, interrupts fromthat source will not be serviced.

A simple example

The following example patches a high level interrupt service routine (ISR) onto the timer/counter 0 interrupt of the RTX. To try this example you must cross-compile the files INTERUPT.FTH and VECTORS.FTH onto your target.

The timer ISR

The example ISR increments a variable which can be fetched in the foreground to detect that the timer is working.

```
VARIABLE TICKS                          \ timer variable

: TICKS-ISR   \ — ; Increment variable
  1 TICKS +!                    \ Increment ticks
;
```

Patching your ISR onto the timer

The ISR needs to be patched onto a timer interrupt. This is made simple as all that is required is that you assign your ISR to be the action of the deferred word TIMER0-ISR.

```
ASSIGN TICKS-ISR TO-DO TIMER0-ISR
```

Once this is done, the timer is ready to go. All that is required is for the timer to be initialised.

Initialising the timer

The timer needs to be initialised to:

- enable Timer 0 interrupts
- enable interrupts globally

To do this a word **INIT-TICKS** is defined:

```
: INIT-TICKS \ — ; initialise the timer overflow
  TIMER0 UNMASK                         \ enable Timer 0 interrupts
  EI                                    \ enable interrupts globally
;
```

The timer can be initialised by typing **INIT-TICKS**.

Testing the timer is running

The timer can be tested by checking the variable **TICKS**:

```
TICKS ?
```

This displays the current value of **TICKS**.

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.

MASK mask —
"mask"

The interrupt(s) specified in the supplied mask are turned off via the Interrupt Mask Register (IMR).

```
swi MASK                \ Disable software interrupts
```

MASK-ALL —
"mask-all"

Turn off all interrupts via the Interrupt Mask Register.

MASKED? mask — t/f
"masked-query"

Returns TRUE if the supplied interrupt(s) are currently disabled via the Interrupt Mask Register

RESTORE-INT sr md cr —
"restore-int"

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

SAVE-INT — sr md cr
"save-int"

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

UNMASK mask —
"un-mask"

The interrupt(s) specified in the supplied mask are turned on via the Interrupt Mask Register (IMR).

```
psv UNMASK              \ Enable data stack overflow ints
```

Blank Page

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 either an angle in 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.

ROM PowerForth Utilities

Supplied as source are utilities to:

- compile source code on your target board via the serial line
- 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 directly 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 the XShell manual.

The required files

To compile text files on your target board, cross compile the files IO-DEF.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 repeatedly 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 inclusive 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:

UPTO end-page

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

For example,

UPTO 10 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:

start-page ALONE

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

For example,

10 ALONE

compiles page ten of the default text file.

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 text file's name. This is normally quicker if you are repeatedly 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-page end-page THRU

This compiles the screens from start-page to end-page inclusive 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 page 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
- a 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 binary image download

Binary 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 binary download utility

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

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

where addr is the start address and #bytes is the number of bytes to download 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 can contain an interactive Forth with the ability to develop an application.

It is recommended that the RAM target code be used to generate the ROM PowerForth EPROMs as this will simplify the hardware requirements.

The notes below describe a ROM PowerForth system built using the RAM target code.

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 two areas of memory:

- one for EPROM
- one for RAM

EPROM area

The EPROM area contains the development kernel, and later, once the application has been developed, the application itself.

RAM area

The RAM area is used to hold a copy of the ROM image plus the application code under development. Once the application is complete, this entire area is down-loaded to the PC, and blown into EPROM, ready to be fitted to the target board in place of the kernel EPROM.

Making your application turnkey

Once your application has been developed, it needs to be made turnkey so that it starts executing at powerup. The application can be made permanent by copying the image into an EPROM.

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
```

Note that if you are cross compiling your application you should use **MAKE-TURNKEY** as described in the “Generating a ROM Target”and “Generating a RAM Target” sections of this manual. It is recommended that **SETUP** only be used where a cross compiler is not available.

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.

Downloading a target image

You can download a target image to your PC using the BIN-DOWN utility supplied. You will require a suitable communications package providing an XMODEM download facility (e.g. Procomm)

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

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 download starting from addr. This should be your entire RAM area

For example, if your RAM area extends from 0000 through 7FFF you would use

```
hex
00000 07FFF BIN-DOWN
```

This will produce a 32K image which can be blown into EPROM and inserted in place of the kernel EPROM.

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$. e.g.

“” 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. e.g.

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. e.g.

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

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

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. e.g.

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. e.g.

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

USING \ — ;
"using"

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

Paged targets

Many people develop for 8-bit and 16-bit target processors. One disadvantage of using 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.

The RTX processor has a built in paging mechanism which allows up to 16 64K pages, any one of which may be in memory at any time.

An example memory map for the MPE Powerboard is shown figure 12.

- Boot ROM - This contains the Forth kernel in EPROM. When the board is powered up it copies itself into RAM and executes from there. It maps into page 0
- RAM - In this example, there are two pages, 0 and 1. Page 0 contains the Forth kernel as copied out from ROM. Page 1 may contain any other data or code that the user chooses to put there. When compiling code, it is not necessary to enter extra code to switch pages. The compiler will do this for you. However to

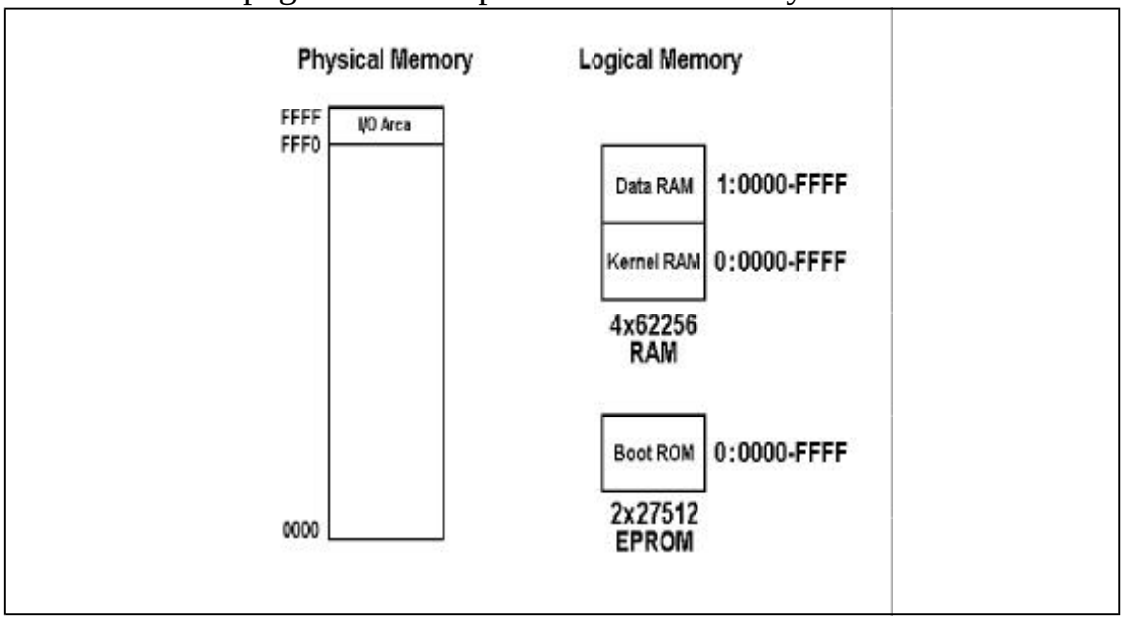

Figure 12 - Example paging mechanism

fetch data from another page it is necessary to specify which page the data is in. You can do this by using the long fetch and store words provided.

- I/O Area - The MPE RTX Powerboard's access to the outside world. This is mapped into every page and exists between F000h and FFFFh.

Note that other boards may use other paging mechanisms to the one described.

Since Forth requires repeated access to the kernel, it may often be more efficient to compile the kernel into all code pages than compile inter-page calls.

Creating a paged target

To create a paged target, you need to define each page's memory map. The actual switching is built in to the processor, and the compiler will automatically produce inter-page calls where necessary. Thus you only need to specify the page number (0-15) that you wish to use.

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
- the page number you wish to use

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-no> CODE-PAGE <page-name>

DATA-PAGE is used in the form:

<start> <end> <page-no> DATA-PAGE <page-name>

where <start> is the start address in the page, <end> is the last address in the page and <page-no> is the page number you are defining. This is used to in-

dicate which page to switch to. The <page-name> is an identifier for the page, and in the case of code pages defines the name of the image file that will be produced.

For example, to define the three pages in figure 12 (RAM target) you would code:

```
$0000 $EFFF  KERNEL Kern
$0000 $0000 0 DATA-PAGE dummy
$0000 $EFFF 1 CODE-PAGE Page1
```

if page 1 contained code, and

```
$0000 $EFFF  KERNEL Kern
$0000 $0000 0 DATA-PAGE dummy
$0000 $EFFF 1 DATA-PAGE Page1
```

if page 1 contained data.

Any other pages are coded in a similar way. The page-number used is the same as the page that you wish to use.

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.

Compiling into a page

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

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 <name>

Where <name> is the name of the page to finish. The cross compiler shows a compilation summary for the page.

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 (RAM Target) or the KERNEL-RAM page (ROM Target) will not be initialised. This must be done at startup by the application.

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 cross compilation
- when to stop cross compilation
- whether to align code to even/odd bytes
- whether to enable floating point
- whether to turn the compiler log on or off
- which code and data page to compile into
- how to 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-compilation, use **FINIS**. **FINIS** stops cross-compilation, closes all files and returns to XShell.

Aligning generated code

The RTX family processors require instructions to be word aligned. To instruct the compiler to do this use the directive **ALIGN.**

Enabling floating point

If you want to 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 what page code and data is 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 **KERNAL-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 figure 13 and 14. The examples given perform conditional compilation inside and outside a colon definition.

Conditional compilation outside a colon definition

The example shown in figure 13 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.

```
1 EQU 1OR2?


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

Figure 13 - Conditional compilation example (1)

Conditional compilation within a colon definition

Using conditional compilation within a colon definition is slightly more compilcated. 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 exam-

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

Figure 14 - Conditional compilation example (2)

ple is shown in figure 15, where a constant **3OR4?** is added to the compiler. This can then be used in the example in figure 14.

```
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
```

Figure 15 - Adding words to the compiler

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 provides more information if you are interested or want to perform more advanced modifications to the cross-compiler or target.

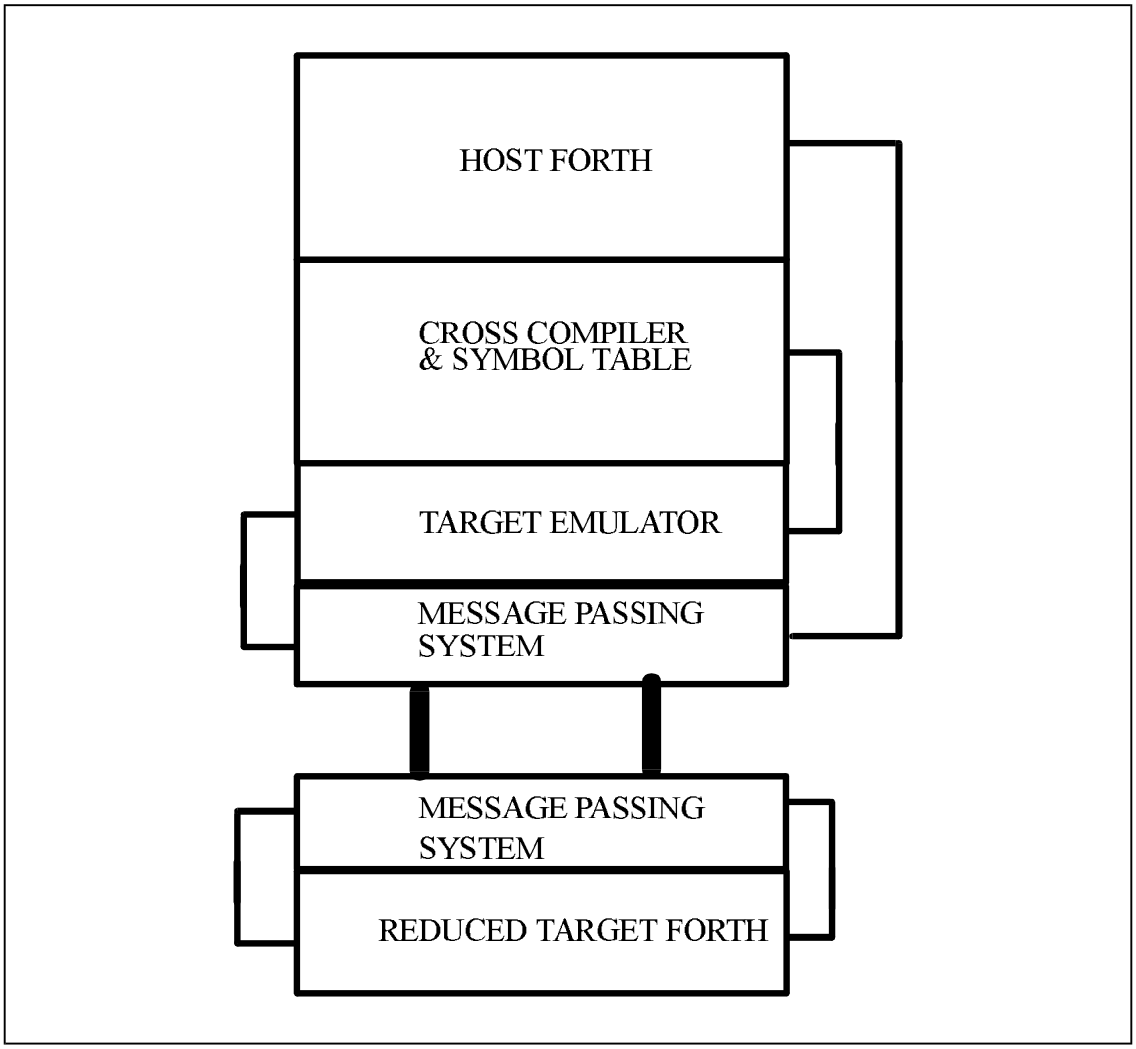

Figure 16 - Umbilical forth message passing

Inside Umbilical Forth

Umbilical Forth behaves in the same way as the ROM and RAM target Forths, but the internal mechanism is totally different. 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 basic cross-compiler.

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 and its page number. By using a message passing system between the cross-compiler and the target, the CFA and page of the word can be passed to the target. The target 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 as you are not communicating with the target directly.

Inside a ROM and RAM target Forth

A 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 characters 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.

Optimising 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 faster you can go round this cycle, the sooner 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 after the word **CROSS-COMPILE** and any macros must be loaded.

Note: The old image file is used by the compiler. This must exist in the compilation directory.

An example

An example control file can be found in the directory RAM/PARTIAL.

Speeding up the downloading

The cross compiler has the facility to download the compiled 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 for
- the bus width (e.g. 8 bit, 16 bit)
- which page to put in the emulator

You will ablso need the correct emulator TSR driver installed in your machine. This should be TSR021 for LePROM emulators and TSR041 for LeMeg and LeBig emulators.

Note: This facility cannot be used with partial compilation.

Setting the 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 17 and 18.

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

Table 17 - Available bus widths

EPROM type	size
2764	e2764
27128	e27128
27256	e27256

Table 18 - Available EPROM sizes

For example, if your board uses a 27256 and your target has a 16-bit bus width, code:

e27256 16bit 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.

Blank page

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 that on which it is running.

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 its 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 to generate an 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.

RAM target Forth A Forth which on power up copies itself from ROM to RAM and then executes from RAM.

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

ROM/RAM target A target board 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.

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. Therefore, if you want to use your old control file you need to modify the way:

- the memory map is defined
- an EPROM emulator is used
- code is compiled into a page

Defining the memory map

The memory map in version 4 control files is defined as in figure 17. The equivalent v5 memory definition is shown in figure 18. 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** is the same as in version 4.

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 ROMRTX to an EPROM emulator starting at address 0000h, you code:

```
$0000 IN-EMULATOR ROMRTX
```

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
\ The ROM for RTX systems is nearly always at 0000h
00000 ROM-BASE                \ ROM starts at 00000
\ 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
0F000 MEM-END                 \ RAM ends at xxxx-1
```

Figure 17 - Example version 4 memory definition

```
$0000 $7FFF    KERNEL ROMRTX                    \ Define kernel ROM
$8000 $EFFF 0  KERNEL-RAM ROMRTX-DATA  \ Define kernel RAM        $F000        MEM-
END                                  \ End of usable memory
```

Figure 18 - Example version 5 memory definition

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, code:

USE-CODE <name>

where <name> is the image's name (i.e. ROMRTX 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. ROMRTX-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 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 files purpose followed by any other general information.

```
\ MPE RTX PowerBoard target control file
pto
Released for use with the MPE Forth Cross Compiler by:

MicroProcessor Engineering
133 Hill Lane
Shirley
Southampton SO1
England

tel: (+44) 703 631441   (international)
     0703 631441        (domestic)
```

Setting the cross-compiler search order

The cross-compiler's vocabulary search order is set so that commands can be found.

```
only forth definitions decimal
```

Loading macros/Opcode definitions

At this stage any macros must be compiled. Macros must be loaded before the command **CROSS-COMPILE**. (see below)

```
all from-file rtx2000.def          \ configure to correct processor
```

Different processors in the RTX family have slightly different opcode sets, so the above line will load in the definitions for the RTX2000.

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 16bit Output-Emulator   \ define EPROM & Width
```

Activating the 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
```

Select the type of target

You must tell the compiler whether you wish to generate code for a ROM or RAM target. The default is for a RAM target.

```
Rom-Target                    \ Generate ROM target code
```

Setting the target's search order

The target's search order must be set. This tells the compiler to compile code on top of the target's Forth vocabulary.

```
only forth definitions
```

Setting the alignment mechanism to be used

If a 16-bit value is to be retrieved, it must be taken from a word aligned (even) address. This restriction forces the compiler to place all instructions on even addresses. To do this, strings must be padded out with an extra space if necessary.

```
align                         \ headers word aligned
```

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.

```
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**.

```
\ memory definitions
$0000 $7FFF    KERNEL ROMRTX                    \ Define kernel ROM
$8000 $A000 0  KERNEL-RAM ROM-DATA              \ Define kernel RAM
```

Output into EPROM emulator

The cross-compiler can send a target image directly to an EPROM emulator, which removes from the cycle 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 to. The following example sets the compiler to download the image ROMRTX, starting at address 0000h.

```
$0000 IN-EMULATOR ROMRTX  \ 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 compila-

tion pages can be set throughout the code, so redirecting the code to different pages.

```
use-code ROMRTX                 \ Select code page
use-data ROM-DATA               \ Select data page
```

Configuring for ROM PowerForth

If the ROM PowerForth utilities are being loaded, the start of the application RAM/ROM area must be defined.

For example, if the application area is the upper half of a 16k of battery backed RAM, you would code:

```
0A000 equ appl-rom
```

Setting the Clock Speed and Baud Rate

The compiler needs to know how fast you wish the serial line to run. To calculate this is also needs to know the crystal speed of your board. It is recommended that for new designs you use 2400 Baud until you are satisifed that the Forth is fully functional on your board.

```
10000 equ clock-khz
38400 equ serial-baud
```

Setting up the interrupt vectors

The RTX family of processors allow you to move the interrupt vector area in memory. This means that you must set this up initially.

```
0 equ INT-BASE \ interrupt vector base - must be 1KB aligned
```

Setting the stack size

The on-chip stack size of the RTX processor in use needs to be specified.

```
$80 constant stack-size
```

Defining the number of tasks

In a multitasking target the number of tasks need to be set. Each task takes up 256 bytes of RAM, so a full 8 tasks takes up 2k 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
                                  \ each task needs 0100 bytes
                                  \ this space is reserved first
```

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 per-task                \ size of each tasks user area
```

Calculating the total memory requirement

The total RAM required by the system is given by the equate **US.** This is the amount of memory required for one task multiplied by the number of tasks in the system.

```
#tasks per-task * Equ Us          \ space used for task pages
```

Compiling the kernel

The main source code which makes up the interactive Forth kernel is now compiled.

```
decimal  all  from-file startup                 \ start up code
decimal  all  from-file code2000                \ main code defs.
decimal  all  from-file kernel                  \ Forth high level kernel
decimal  all  from-file drivers\pbrdio          \ UART driver code
decimal  all  from-file rtx-tool                \ Toolkit
decimal  all  from-file ucodertx                \ RTX Opcodes
```

Compiling the multitasker

The multitasker source will only be compiled if the number of tasks specified is greater than one.

```
#tasks 1 >
if(
  decimal  all  from-file multirtx              \ multi-tasker
)endif
```

Compiling the software floating point

The software floating point consists of two files, SOFTFP.FTH, and either FPRAM.HI or FPROM.HI depending on whether a RAM or ROM target is in use. It will only be compiled if the SOFTFP? flag was set earlier in the control file.

```
\ Software floating point
softfp?
if(
  decimal  all  from-file softfp\fpromhi.fth    \ ROM target 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 hard disk services from the target system. It will only be compiled if the ROMFORTH? flag was set earlier in the control file.

```
\ 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 RTX2000/1A/10 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 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
```

Blank Page

Appendix C
Error Messages

Error messages are kept in the screen file ERTX.XS3 in the COMPILER directory. Error numbers start at 0, and each error number refers to a line starting at line 0. This format allows the error message file to be maintained using any screen file editor.

The error messages are listed in different categories:

- general Forth errors
- system messages
- 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 9 - cannot compile from screen zero. Screen 0 should be used for comments only.

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 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.

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 58 - Bad module version code - recompile using correct SLAVE.xxx. This is an internal software error. Contact MPE.

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 read error.

Error 68 - screen file write error.

Error 69 - path not found. Usually because the file or path name has been misspelled.

Error 70 - starting screen number less than ending screen number.

Error 72 - Memory buffer release error

Error 73 - Memory buffer allocation error

Error 74 - Source file nesting level too deep

Error 75 - No source file to un-nest

Error 76 - End of file before requested page

Error 77 - Screen file close error

DOS errors 80..112

Error 81 - invalid funtion 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 - not enough RAM or memory fragmented.

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.

Error 122 - missing right bracket - must be space separated.

Blank Page

Appendix D
Technical Support

Technical Support

Technical support is available from MPE by fax or phone during office hours (9am-5pm UK Time). If reporting a problem, please fax a short piece of code which illustrates it, so that we can respond to you quickly. You can also obtain technical support via email, or by access to our own technical support conference on the CIX(Compulink Information eXchange) bulletin board system.

tel: +44 703 631441
fax: +44 703 339691
cix (voice): +44 81 390 8446
cix (bbs): +44 81 399 1244
Internet: mpe@cix.compulink.co.uk

Blank page

Index

V

X